AUTOMATED SUPPORT FOR IMPROVING SOFTWARE QUALITY OF MOBILE APPLICATIONS BEFORE AND AFTER RELEASE

A Dissertation
Presented to
The Academic Faculty

By

Mattia Fazzini

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computer Science

Georgia Institute of Technology

August 2019

AUTOMATED SUPPORT FOR IMPROVING SOFTWARE QUALITY OF MOBILE APPLICATIONS BEFORE AND AFTER RELEASE

Approved by:

Dr. Alessandro Orso, Advisor
School of Computer Science
Georgia Institute of Technology

Dr. Wenke Lee
School of Computer Science
Georgia Institute of Technology

Dr. Vivek Sarkar
School of Computer Science
Georgia Institute of Technology

Dr. Tevfik Bultan
Department of Computer Science
University of California, Santa Barbara

Dr. Denys Poshyvanyk
Department of Computer Science
College of William & Mary

Date Approved: June 17, 2019

To my wife,

Mariko.

To my parents,

Maria and Gilberto.

ACKNOWLEDGEMENTS

I would like to thank many people whose help, support, and feedback were instrumental for developing this dissertation.

First and foremost, I would like to thank my advisor, Dr. Alessandro Orso. He gave me the opportunity to pursue this Ph.D. degree and mentored me on how to conduct high-quality research. His work ethic has always been inspiring, which motivated me every day to be a better researcher. In addition, throughout my studies, he also helped me on several personal aspects for which I will always be grateful.

I would also like to thank my committee members, Drs. Tevfik Bultan, Wenke Lee, Denys Poshyvanyk, and Vivek Sarkar, for all of their support in improving this dissertation. They also provided invaluable help to continue my passion for research in the future.

My studies would have not been the same without my labmates. They gave me a great sense of community that I will carry with me for the rest of my life. I would like to thank Dr. Wei Jin as he played a key role in the earlier part of my studies.

I would also like to thank my friends Robert Conroy, Graham Parkinson, and Will Thies as they helped me in finding fresh energy when needed.

My Ph.D. studies would have not been possible without the support from my parents Maria and Gilberto, and my grandfather Vinicio. The examples provided by my parents have taught me to work hard for the things that I aspire to achieve. I will be always grateful that they encouraged me in pursuing my interests.

Finally, I would like to thank my wife Mariko for her continuous support during my studies. This dissertation would not have been possible without you. It is very hard to put into words my gratitude for having you in my life.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Mobile devices are becoming the platform of choice for a number of tasks, including news consumption, online shopping, and streaming content. Mobile applications (or simply apps) play an essential role in the success of mobile platforms and impact our lives fundamentally. To improve the quality of these apps before release, companies invest a great amount of resources in software verification, and in particular in testing. It is therefore crucial to define and use testing approaches that are both effective and efficient. At the same time, because testing techniques cannot generally reveal all bugs, companies release apps containing latent bugs. Additionally, the environment in which apps operate changes quickly and these changes introduce new bugs. The ability to react effectively to reported latent bugs and changes to the environment is therefore also essential to resolve bugs, but the support for these tasks is still limited and based on mostly manual, human-intensive approaches. The overarching goal of this dissertation is to improve software quality by devising automated testing and maintenance techniques that address these problems.

To this end, I defined a family of testing and maintenance techniques: BARISTA, DIFFDROID, YAKUSU, and APPEVOLVE. BARISTA records, encodes, and runs platform-independent test cases to help developers in testing apps. DIFFDROID identifies inconsistencies in the behavior of an app running on different platforms and reports these inconsistencies to developers. YAKUSU translates natural-language bug reports into test cases, so that developers can use the generated tests to debug failures caused by latent bugs and quickly fix their apps. APPEVOLVE accounts for bugs caused by changes to the environment; it does so by automatically updating API usages (i.e., interactions with the underlying environment) in an app based on how developers of other apps performed corresponding changes.

To evaluate the effectiveness of my techniques, I implemented them as prototype tools and performed a series of empirical investigations on real-world apps. The evaluation shows that the techniques are not only effective but also efficient. These results provide

evidence that the techniques can be practically used to improve software quality of mobile apps before and after release.

CHAPTER 1
INTRODUCTION

Software is present in many facets of our lives. It is in the things we use every day (*e.g.,* smartphones, laptops, and cars) and it is also part of the infrastructure of our society (*e.g.,* schools, hospitals, and airports). It is indeed not easy to think about a device, system, or organization that provides some complex functionality and does not rely on software. This situation also means that we are increasingly dependent on software. For this reason, it is imperative to have and use techniques that improve software quality. Specifically, we must have and use techniques for improving software quality before and after software release.

We must employ techniques to improve software quality before release because we want to use functioning software. Not using these techniques is always problematic and in many cases can have catastrophic consequences. The launch of the *HealthCare.gov* website in 2013 provides an example of such negative consequences. The website was created to provide individuals with a marketplace to compare and select health insurance plans. When the website was released, it contained a large number of bugs that prevented millions of people from accessing health insurance services [162]. To understand the reason behind the presence of these bugs, the House committee interviewed the former United States secretary of health and human services, and the secretary admitted that the software should have been tested better [137].

At the same time, because testing and other software verification techniques (used before releasing software) cannot generally reveal all bugs, and because software is always evolving, it is also imperative to improve software quality after release. Also in this case, not doing this task properly has negative consequences on the software users. For example, a bug in *Whatsapp*—a messaging application—was deleting messages from users' conversations [26, 141]. Unsurprisingly, the users affected by the bug were extremely upset.

Moreover, they were particularly upset because, to add to their frustration, the bug was not fixed in a timely fashion despite the number of times they reported the issue.

The examples mentioned above are just two of the many cases [167] that highlight a strong need for automated techniques to improve software quality.

1.1 Thesis Overview

In particular, in this dissertation work I focus on mobile applications (or simply apps) due to their popularity and the potential for practical impact associated with the research. In fact, mobile apps are becoming the most popular software [20]. We use mobile apps to perform a large number of daily activities, including listening to music, reading books, and ordering food. Given the impact that apps have on our lives, it is a priority to use automated testing and maintenance techniques for improving the quality of apps before and after release. Unfortunately, current support provided by these types of techniques is still limited and based on manual, human-intensive approaches. This situation presents us with important research problems and challenges that need to be addressed.

1.1.1 Testing Challenges

Generating Test Cases

Apps must be tested to gain confidence that they behave as expected. This is especially important for mobile apps as they are part of a highly competitive market and a failure in an app can negatively impact the reputation for the company that provides the app. In fact, an astonishing 88% of app users would consider abandoning an app if they encountered bugs or glitches [21]. Unfortunately, testing of mobile apps is today mainly performed manually [100], as app developers report that they lack effective methods and tools to do otherwise. Specifically, a study involving 1660 companies from 32 countries [25] reports that 52% of the companies do not have enough time to test, 47% of them do not have the right testing process or method, and 46% of them do not have the right tools to test.

Analyzing Cross-Platform Software Behavior

In the case of mobile apps, the task of establishing whether the software behaves as expected is further complicated by the fact that apps must run on platforms[1] with different properties; thus, developers are faced with the problem of checking that the apps' behavior is consistent across platforms. This problem is particularly relevant in the context of Android [62] apps, as their ecosystem is heavily fragmented. In fact, given the open source nature of Android and vendors' intent to satisfy different market needs, platforms running Android are often customized (in terms of both their hardware and software), leading to a multitude of platforms concurrently available in the field. For instance, the number of distinct platforms in the year 2015 was $24,093$ [132]. The differences in the platforms can lead to inconsistencies in the behavior of an app, which in turn can affect the functionality of the app.

1.1.2 Maintenance Challenges

Analyzing Bug Reports

Because app development is driven by demanding time-to-market constraints [158], and because testing and other software verification techniques cannot generally reveal all bugs, it is virtually impossible for developers to identify all bugs before releasing an app. Consequently, it is common for users to experience failures. When users experience a failure in an app, they can submit a bug report to developers to provide information on how the failure happened. Developers investigate reported failures, find their causes, and eliminate them. To do so, developers must look at a bug report, understand what steps led to the reported failure, try to reproduce these steps and the corresponding failure, and debug the failure. Performing these tasks can be extremely time-consuming [164], especially in the presence of a large number of (often incomplete) reports [18, 19].

[1] Hereafter I use *platform* to refer to the combination a mobile device's hardware and operating system.

Supporting Changes to the Software Environment

After releasing apps, developers also need to account for changes to the environment in which apps operate, as these changes can affect the behavior of their apps. One of the most frequent type of these changes is the update of the application programming interface (API) offered by the platform [111, 98, 17, 70, 178]. Changes to the API can significantly affect the behavior of apps, as apps rely heavily on the API to access a number of services without which they could not operate. The task of updating an app is tedious and time consuming, as the number of changes can be large and spread across the app code. This situation is even more problematic for Android apps as the fragmentation of the ecosystem requires apps to be compatible with multiple versions of the API.

1.2 Approach

To address the challenges I just discussed, I (1) defined a family of testing and maintenance techniques BARISTA, DIFFDROID, YAKUSU, and APPEVOLVE, (2) implemented these techniques to support Android apps, and (3) evaluated the techniques on real-world apps. In the rest of this section, I provide an overview of these techniques and their evaluation. It is worth noting that, although the techniques are defined in the domain of mobile apps, the principles behind them are general. There should therefore be no major barrier to extending my techniques to other domains (e.g., web applications).

1.2.1 Generating Test Cases

BARISTA [38] is a testing technique for generating test cases based on record and replay. Current record-and-replay based solutions [46, 84, 50] generate test cases that are brittle and break while running on platforms other than the one on which they were recorded. In addition, they have limited support to help developers define how the software should behave. BARISTA records, encodes, and runs platform-independent test cases. The technique

is characterized by the following aspects. First, the technique implements the record-once-run-everywhere principle. In fact, developers can record their test cases on one platform and rerun them on other platforms. To do so, the technique analyzes a tree representation of the user interface (UI) while recording and executing test cases. Second, the technique allows developers to create oracles in an intuitive way by directly interacting with the UI without affecting the functionality being tested. Third, the technique is minimally intrusive as it does not require modifying the operating system, making BARISTA applicable to any platform. The evaluation of the technique shows that BARISTA successfully encodes and executes 46% more test cases than the state of the art. In addition, BARISTA users recorded test cases 32% faster than the baseline. Finally, developers used BARISTA in a company environment and acknowledged its effectiveness.

1.2.2 Analyzing Cross-Platform Software Behavior

DIFFDROID [39] is a testing technique that analyzes cross-platform software behavior. Specifically, it identifies cross-platform inconsistencies (CPIs) when an app is running on different platforms. DIFFDROID identifies CPIs that manifest through an app's UI. The technique combines input generation, UI modeling, and differential testing, and it operates in four steps. First, given an app running on a reference platform, the technique performs random-input generation to exercise the app's functionality. Second, DIFFDROID builds a reference UI model based on the states explored during input generation. Third, the technique executes the same inputs on a set of test platforms to collect a test UI model for each platform. Fourth, the technique compares the reference UI model with each test UI model and reports any differences between the models as CPIs. The comparison leverages a tree representation of the UI to find missing elements in the UI and uses computer vision techniques to identify visual differences in how elements are displayed. In the evaluation, DIFFDROID identified 96 CPIs for five benchmarks running on 130 different platforms.

1.2.3 Analyzing Bug Reports

YAKUSU [40] is a maintenance technique that analyzes natural language bug reports and translates them into test cases. With these test cases, developers can immediately focus on debugging their apps. The technique uses a combination of static analysis, dynamic analysis, and natural language processing techniques. The task of translating natural language bug reports into test cases contains three principal challenges. First, extracting actionable information from natural language is a non-trivial task, as it involves interpreting imprecise and context-dependent descriptions. Second, a logical gap can exist between such steps and UI events used in a test case. Third, the sequence of steps may be incomplete. The technique overcomes these challenges using a three-step approach. First, YAKUSU statically analyzes the relevant app to compute an ontology of the UI elements that the app uses. Second, the technique analyzes sentences in a bug report, processing their dependency tree representation, then translates them into a sequence of abstract steps with the help of the ontology. Finally, YAKUSU dynamically explores the relevant app to map abstract steps to executable UI actions and encodes them into a test case. In the evaluation, YAKUSU successfully generated a test case from a bug report in 59.7% of the cases considered.

1.2.4 Supporting Changes to the Software Environment

APPEVOLVE [41] is a maintenance technique that accounts for changes to the software environment caused by updates to the API. APPEVOLVE automatically updates API usages (i.e., any call of one or more methods to the API) in an app by analyzing how developers of other apps performed corresponding changes. Automating this task is particularly challenging as updates from different developers do not always share the same set of operations. APPEVOLVE overcomes this challenge and automatically updates API usages in a target app using a four-step approach. First, the technique analyzes the target app to identify API usages that API changes affect. Second, APPEVOLVE searches existing code bases for examples of updates that are also compatible with previous API versions. Third,

the technique analyzes update examples to identify the core of the update, ranks the examples based on their proximity to the core, and transforms the examples into generic update patches. Finally, APPEVOLVE applies the generated patches to the target app in order of their ranking, while performing differential testing to validate the update. In the evaluation, the technique updated 85% of the API usages affected by changes in a set of randomly selected real-world apps and automatically validated 68% of the updates performed.

1.3 Thesis Contributions

In summary, my dissertation provides the following contributions to the software engineering body of knowledge:

- BARISTA, a technique to record, encode, and run platform-independent test cases.
- DIFFDROID, a technique to detect cross-platform inconsistencies.
- YAKUSU, a technique to translate bug reports into test cases.
- APPEVOLVE, a technique to perform API-usage updates.
- Prototype tool implementations of the techniques.
- Empirical evaluation of the techniques on real-world mobile benchmarks.

1.4 Thesis Organization

The rest of this dissertation is organized as follows. Chapter 2 presents background information on mobile app terminology. Chapter 3 provides an overview of how the techniques presented in this dissertation fit in the app development process and how they can be applied before and after release. Chapters 4, 5, 6, and 7 provide details on BARISTA, DIFFDROID, YAKUSU, and APPEVOLVE, respectively. Chapter 8 describes related work and Chapter 9 provides a conclusion to the research and discusses potential future work.

CHAPTER 2
BACKGROUND

This chapter provides background information and terminology related to mobile apps. This chapter also reports background information and terminology specific to the Android platform[1] and Android apps. The background information and terminology introduced in this chapter is used throughout the remainder of this dissertation.

2.1 Mobile Applications

A mobile app is a software program that execute on top of a mobile platform such as a smartphone or a tablet. Users provide inputs to an app primarily through its UI, which is displayed on the touchscreen of the platform. The UI of an app is a rich, graphical user interface composed by UI elements. A UI element is a graphical component with a specific interface function (*e.g.,* a label displays text, a text box is filled with textual input, and a button triggers a computation). The output computed by an app is generally displayed back to the user through the UI. Apps can also receive inputs from other sources such as a sensor (*e.g.,* accelerometer, gyroscope, and barometer) or a camera. To process inputs from this set of diverse sources, the operating system uses an event-driven model, which apps need to follow. An app registers for events using callbacks that the operating system provides through its API. This API is one part of the software stack composing the operating system on which apps execute. This software stack usually contains a kernel that menages the application framework, which apps interface through the provided API.

[1] Hereafter I use *Android platform* and *Android operating system* interchangeably.

2.2 Android Platform

The Android platform is an operating system created for mobile devices. The platform is a linux-based software stack [65] that includes: (i) a Linux kernel, (ii) an hardware abstraction layer used to interface with the hardware devices on the mobile platform, (iii) an application runtime on which apps execute, and (iv) an API framework that apps use to access a number of services and resources provided by the platform.

The Android platform is in continuous evolution and so is its provided API [111, 98, 17, 70, 178]. For this reason, there are multiple versions (or levels) of the platform available in the field. Because apps interface the platform through its API, and this API might be different in different versions of the platform, apps are allowed to specify on which versions of the platform they can be run on. Specifically, the platform allows an app to indicate the minimum API version (`minSdkVersion`) required by the app to run, and the API version (`targetSdkVersion`) that the application targets (*i.e.,* the latest version of the platform for which the app was designed).

2.3 Android Applications

An Android app [63] is a mobile app running on the Android platform. An Android app is composed by different parts: *app components*, a *manifest* file, and *resources*. To create an executable app, app components, manifest file, and resources are built and packaged together into an Android *package* (or simply *APK*). To run the app, the APK must be installed on a mobile platform running the Android platform.

2.3.1 App Components

App components are the building blocks of an Android app. Each component is an entry point in the app. These entry points can be used by either the Android platform or users to enter the app. There are four types of components: *activities*, *services*, *broadcast receivers*,

and *content providers*. App components use the API [56] of the Android platform to access the features (*e.g.,* network) provided by the platform.

Users can interact with an app through activities. An activity represents a single screen in the app and defines the UI of the screen. Developers create activities through code. The UI of the activity is defined through a combination activity code and *layouts*. A layout is a resource of the app and defines the structure of the UI by combining UI elements (*e.g.,* buttons, labels, and text boxes). Activities are independent from one another but, as a whole, they define how user navigates within an app. Transitions between activities are made through *intents*. Specifically, intents are messages used to activate and enter a specific component in an app. Each app has a special activity called the "main" activity. This activity is the first activity presented to the user when the app starts. Services are components that run in the background and perform long-running operations. This type of component is also used to expose some of the app functionality to other apps. Broadcast receivers are components used to perform computation outside of the regular user flow. These components are activated using messages (*i.e.,* intents). These messages can be received from other components of the app or the Android system. Finally, content providers manage app data and can store this data in the file system, in a database, or on the web. Apps can use a content provider to query or modify the data.

2.3.2 Manifest

An app uses the manifest file for multiple purposes: communicating to Android platform the permissions (*e.g.,* access to user's contacts) required by the app, declaring the components of the app, specifying the minim version of the Android platform on which the app can run, and listing the external libraries used by the app.

2.3.3 Resources

An app uses resources to define anything relating to the visual presentation of the app (*e.g.,* menus, animations, images). App code can reference a specific resource using its identifier. This identifier is called *resource ID*.

CHAPTER 3

OVERVIEW

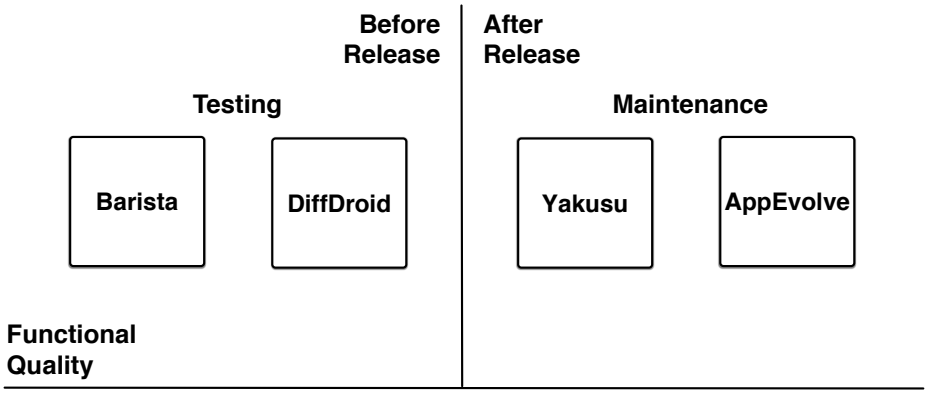


Figure 3.1: Overview of how my techniques fits in the app development process.

Figure 3.1 provides a high level view of how my techniques fit in the app development process and their applicability before and after software release. All of my techniques aim to improve the functional quality of software. BARISTA and DIFFDROID are testing techniques and should be used before software release. BARISTA generates platform-independent test cases through record and replay. DIFFDROID targets cross-platform testing and identifies inconsistencies in the behavior of an app across platforms. YAKUSU and APPEVOLVE are maintenance techniques and should be used after software release. YAKUSU helps with the task of corrective maintenance by translating natural language bug reports into executable test cases. APPEVOLVE helps with the task of adaptive maintenance by performing API-usage updates to account for changes to the API (*i.e.,* the environment in which apps operate.) All of my techniques can be used multiple times in the life of a software system when adopting an iterative development process. Chapters 4, 5, 6, and 7 describe the techniques in detail.

CHAPTER 4
GENERATING PLATFORM-INDEPENDENT TEST CASES

This chapter presents my technique to generate platform-independent test cases for improving software quality before release. The technique, named BARISTA, records developer's interactions during manual testing and generates a platform-independent test case that can be reused to test the app as necessary. In addition, the technique also offers an intuitive way to define test oracles directly on the device. Generated tests are platform-independent as they can run on platforms that are different from the one in which they were recorded. The rest of this chapter is structured as follows. Section 4.1 details BARISTA and Section 4.2 discusses the evaluation of the technique.

4.1 Technique

Figure 4.1 provides a high-level overview of the technique, which consists of three main phases. In the *test case recording phase*, the user interacts with the application under test (AUT) with the goal of testing its functionality. The technique records user interactions together with user induced system events and offers a convenient interface to define assertion-based oracles. At the end of the recording phase, the technique enters its *test case encoding phase*, which translates recorded interactions and oracles into test cases that are (as much as possible) platform independent. Finally, in the *test case execution phase*, the technique executes the encoded test cases on multiple devices and summarizes the test results in a report. In the remainder of this section, I describe these three phases in detail.

4.1.1 Test Case Recording

In this phase, the user records test cases by exercising the functionality of an app. This phase receives the package name of the AUT as input. Based on the package name pro-

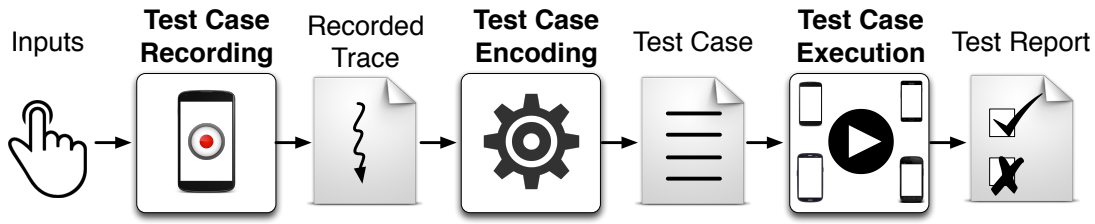


Figure 4.1: High-level overview of BARISTA.

vided, the technique launches the app's main activity [11] and, at the same time, creates an interactive *menu*. The menu is displayed as a floating menu above the AUT and is movable, so that it does not interfere with the user interaction with the app.

As soon as the app is launched, a second component starts operating: the *recorder*. This component is used to (1) access the UI displayed by the AUT, (2) process user interactions, (3) process system events induced by user interactions that affect recorded test cases, and (4) assist the oracle definition process. The recorder leverages the accessibility framework of the Android platform [11] to accomplish these tasks. The accessibility framework provides access to events generated by the system in response to user interface events (e.g., the click of a button). The recorder leverages the accessibility infrastructure to listen to two categories of events: events that describe a change in the UI and events that are fired as consequence of user interactions. Events in the former category are used to create a reference that uniquely identifies an element in the app's UI. I call this reference the *selector* of the element. Events in the latter category are used to identify user interactions. Recorded interactions use selectors to refer to their target UI elements. Interactions and defined oracles are logged by the recorder in the *recorded trace* in the form of *actions*. When the user stops the recorder, BARISTA passes the content of the recorded trace to the test case encoding phase. In the rest of this section, I discuss the content of the recorded trace, describe how the recorder defines selectors, present what type of interactions are recognized by the technique, and describe the oracle definition process.

$trace\text{-}def$ ::= ***trace*** $main\text{-}activity\ actions$
$main\text{-}activity$ ::= $string$
$actions$ ::= $action \mid action,\ actions$
$action$ ::= $interact\text{-}def \mid sys\text{-}interact\text{-}def \mid ui\text{-}assert\text{-}def$
$\mid af\text{-}assert\text{-}def \mid key\text{-}def$
$interact\text{-}def$ ::= ***interact*** $i\text{-}type\ selector\ timestamp\ i\text{-}props$
$i\text{-}type$ ::= ***click*** | ***long click*** | ***type*** | ***select*** | ***scroll***
$selector$ ::= $resource\text{-}id \mid xpath \mid properties\text{-}based$
$resource\text{-}id$ ::= $string$
$xpath$ ::= $string$
$properties\text{-}based$ ::= $element\text{-}class\ element\text{-}text$
$element\text{-}class$:= $string$
$element\text{-}text$:= $string$
$timestamp$ ::= $number$
$i\text{-}props$ ::= $\mid exprs$
$sys\text{-}interact\text{-}def$ ::= ***sys-interact*** $sys\text{-}i\text{-}type\ timestamp\ sys\text{-}i\text{-}props$
$sys\text{-}i\text{-}type$ ::= ***pause*** | ***stop*** | ***restart*** | ***start*** | ***resume*** | ***rotate*** | ***message***
$sys\text{-}i\text{-}props$ ::= $\mid exprs$
$ui\text{-}assert\text{-}def$ ::= ***ui-assert*** $ui\text{-}a\text{-}type\ selector\ timestamp\ ui\text{-}a\text{-}props$
$ui\text{-}a\text{-}type$ ::= ***checked*** | ***clickable*** | ***displayed*** | ***enabled*** | ***focus***
$ui\text{-}a\text{-}props$ ::= $\mid selector \mid exprs$
$af\text{-}assert\text{-}def$ ::= ***af-assert*** $timestamp\ af\text{-}a\text{-}props$
$af\text{-}a\text{-}props$ ::= $\mid exprs$
$key\text{-}def$ ::= ***key*** $key\text{-}type\ timestamp$
$key\text{-}type$ ::= ***action*** | ***close***
$exprs$ ::= $expr \mid expr,\ exprs$
$expr$ ::= $bool \mid number \mid string$

Figure 4.2: Abstract syntax of the recorded trace in BARISTA.

Recorded Trace

Figure 4.2 shows an abstract syntax for the recorded trace. The beginning of the trace is defined by the $trace\text{-}def$ production rule, which indicates that a trace consists of the name of the main activity followed by a list of actions. The types of actions logged into the recorded trace is indicated by production $action$. In the rest of this section, I will refer to the abstract syntax while describing the actions recorded in this phase.

Selectors

The technique creates a selector for user interactions and oracles, which is used to accurately identify the UI element associated with these actions and is independent from the screen size of the device. BARISTA defines and uses three types of selectors: (1) the *resource ID selector* ($resource\text{-}id$ in Figure 4.2), (2) the *XPath selector* ($xpath$), and (3) the *property-based selector* ($property\text{-}based$). The resource ID selector corresponds to the Android resource ID that is associated to a UI element [11]; the XPath [169] selector

identifies an element based on its position in the UI tree (as the UI tree can be mapped to an XML document); and the property-based selector identifies an element based on two properties: the class of the element ($element\text{-}class$) and the text displayed by the element ($element\text{-}text$), if any.

The technique does not use the Android resource ID as its only type of selector because the Android framework does not require a developer to specify a resource ID for each UI element. Moreover, the framework cannot enforce uniqueness of IDs in the UI tree. BARISTA does not use an element's screen coordinates as a selector either because the screen coordinates of a UI element can be considerably different on different devices.

The recorder aims to identify the most suitable type of selector for every interaction and oracle processed by leveraging the accessibility functionality of the Android platform. It does so by analyzing the accessibility tree for the UI displayed on the device. Each node in the tree represents an element in the UI and is characterized by two properties of interest: resource ID (if defined) and class of the UI element represented by the node. The recorder navigates the accessibility tree to track uniqueness of resource IDs and stores the IDs and the corresponding nodes in a *resource ID map*. The information stored in this map is then used every time an interaction occurs or an oracle is defined by the user. More precisely, when the recorder processes these types of actions, it considers the accessibility node associated with the action. The recorder checks whether the node has a resource ID and, if it does, checks for its uniqueness using the resource ID map. In case the resource ID is unique, the recorder creates a selector of type resource ID for that action. If the node associated to an action does not have a resource ID or the ID is not unique, the recorder generates a selector of type XPath, where the XPath selector is a path expression that identifies a specific node in the tree.

When the window containing the element affected by an interaction becomes inactive immediately after the interaction is performed (*e.g.,* when selecting an entry of a `ListPreference` dialog), the accessibility framework does not provide the reference to

the node in the accessibility tree affected by the interaction. In this case, the recorder cannot define a resource ID or XPath selector and uses a property-based selector instead. The property-based selector leverages the information stored in the accessibility event representing the interaction (see Section 4.1.1). This type of selector identifies an element in the UI using the class of the element and the text displayed by the element (if any). I selected these two properties because they will not change across devices with different screen properties. Two UI elements that belong to the same class and display the same text would have the same selector and would thus be indistinguishable. Although this could be problematic, this type of selector is used only when the resource ID and XPath selectors cannot be used, which is not a common situation and never occurred in the evaluation.

Interactions

The recorder recognizes user interactions by analyzing accessibility events created by the Android platform as a result of such interactions. These events have a set of properties that describe the characteristics of the interactions. I illustrate how the recorder processes two types of events, as other events are handled similarly.

Click The technique detects when a user clicks on a UI element by listening to accessibility events of type `TYPE_VIEW_CLICKED`. The recorder encodes an event of this type as an entry in the recorded trace ($interact\text{-}def$ in Figure 4.2). It labels the entry as of type ***click*** ($i\text{-}type$), identifies the interaction selector ($selector$) as discussed in Section 4.1.1, and saves the action timestamp ($timestamp$).

Type The technique recognizes when a user types text into an app by processing accessibility events of type `TYPE_VIEW_TEXT_CHANGED`. Naively recording events from this class, however, would result in a recorded trace that also includes spurious events in the case of programmatic (i.e., not user driven) modifications of the text. To address this issue, BARISTA leverages the fact that actual typing is always followed by a `TYPE_WINDOW_CON-`

TENT_CHANGED event. For each typing event, the recorder encodes the event as an entry in the recorded trace ($interact\text{-}\ def$), labels the entry as of class ***type*** ($i\text{-}type$), identifies the interaction selector ($selector$), saves the action timestamp ($timestamp$), and adds the text typed by the user to the properties of the entry ($i\text{-}props$). It is worth noting that, when a user enters text incrementally, this results in a sequence of events. This sequence of events is processed in the test case encoding phase to minimize the size of the generated test cases (see Section 4.1.2).

After typing text, a user can click the input method action key to trigger developer defined actions. Because the Android system does not generate accessibility events for this type of interactions, the technique provides an on-screen keyboard that can be used by the tester as a regular keyboard and records this type of interactions as well. In response to this event, the recorder adds an entry ($key\text{-}def$) to its recorded trace (***action***). The technique handles in a similar way the key that, when clicked, hides the on-screen keyboard (***close***).

User-induced System Events

User interactions can lead to system events that affect the AUT and consequently the behavior of recorded tests. I classify these events under three categories: events that trigger callbacks of the activity lifecycle; runtime changes in the configuration of the device; and messaging objects (intents) that originate from other apps and trigger the execution of components in the AUT. I will next illustrate how BARISTA accounts for these events.

Activity Lifecycle Callbacks These are triggered by the Android system as result of certain user interactions and can be divided into (1) callbacks generated as the user navigates through the AUT and (2) callbacks triggered when the AUT is not running in the foreground. The technique does not take any action on callbacks of the former type because they are automatically triggered in the test scripts generated by BARISTA. Conversely, the technique detects and suitably processes the latter type of callbacks. The recorder detects

when an activity of the AUT stops running in the foreground by analyzing accessibility events of type TYPE_WINDOW_STATE_CHANGED. In this case, the recorder checks if the activity is in its PAUSED or STOPPED state by accessing the activity manager running in the Android system. When the AUT starts running in the foreground again (detected by the recorder using the accessibility event mentioned above), the recorder creates entries ($sys\text{-}interact\text{-}def$) of type ***pause*** and ***resume*** ($sys\text{-}i\text{-}type$) if the activity was in the paused state. Otherwise, if the activity was in the stopped state, it adds ***pause***, ***stop***, ***restart***, ***start***, and ***resume*** entries.

Device Configurations Configurations can be changed at runtime by the Android system as result of certain user actions. Among those, screen orientation is particularly important for test case recording because an activity of the AUT can display different UI elements based on the orientation of the device. The technique records such changes so that the test execution phase can properly execute recorded interactions. The recorder listens for configuration change events generated by the Android system and when it detects a screen orientation change it stores the change as an entry ($sys\text{-}interact\text{-}def$) of type ***rotate*** ($sys\text{-}i\text{-}type$) having the current orientation value as its property ($sys\text{-}i\text{-}props$).

Intents Intents are the messaging objects used by the Android system to enable communication between different apps. An app can let the system know about what messages is interested in receiving by using intent filters [11]. The technique allows users to define and send intents to the AUT so that they can test the behavior of the AUT upon receiving these messages. Users can also define the properties of an intent through the menu provided by BARISTA. When an intent is defined, the recorder saves it together with its properties as an entry ($sys\text{-}interact\text{-}def$) of type ***message*** into the recorded trace and then sends the intent to the AUT.

Table 4.1: Assertable properties for UI-based oracles in BARISTA.

Property	*Description*
CHECKED	The element is checked
CLICKABLE	The element can be clicked
DISPLAYED	The element is entirely visible to the user
ENABLED	The element is enabled
FOCUS	The element has focus
FOCUSABLE	The element can receive focus
TEXT	The element contains a specific text
CHILD	Child-parent relationship between two elements in the UI
PARENT	Parent-child relationship between two elements in the UI
SIBLING	Sibling relationship between two elements in the UI

Oracles

Oracles are an essential part of a test case. The technique uses assertion based oracles that can be of two types: UI-based and activity-flow-related oracles. The former check for properties of UI elements, whereas the latter check for properties of intents used to transfer control between AUT components.

UI-based Oracles These oracles can either check the state of a UI element at a specific point of the execution or check the relationship between two UI elements. Table 4.1 reports the properties that can be asserted using UI-based oracles and provides a brief description of them. Variations of the properties listed in Table 4.1 can also be asserted. For instance, the technique can be used to assert that the percentage of visible area of an element is above a user defined threshold. Moreover, BARISTA can also define assertions that check that a property of an element does not have a certain value. The menu and the recorder contribute together to the creation of assertions. Figures 4.3, 4.4, and 4.5 show part of the assertion creation process. The user starts the process by clicking the *assert button* in the menu (the button with the tick symbol in Figure 4.3). This creates the *assertion pane*, a see-through pane that overlays the device screen entirely (Figure 4.4). This pane intercepts all user interactions and is configured so that the Android system does not generate accessibility events for interactions intercepted on the pane, so that no spurious events are recorded.

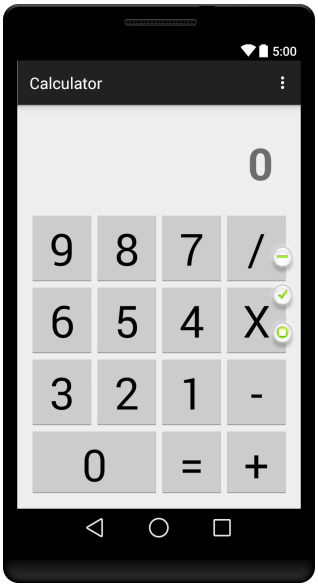


Figure 4.3: Menu overlay in BARISTA.

At this point, the user can define assertions either automatically or manually. With the automatic process, the user selects an element in the UI, and the technique automatically adds assertions for each property of the element. With the manual process, assertions are defined directly by the user. I describe in detail only the manual process as the automatic process follows similar principles.

As shown in Table 4.1, the user can assert properties that affect either a single element or a pair of elements. I illustrate how the technique works when asserting properties that affect one element. (Assertions that affect a pair of elements are defined similarly.) The user selects an element in the UI by long clicking (tap-hold-release) on it. In response to the long click, BARISTA sends the x and y coordinates of the location being pressed to the recorder, which explores the accessibility tree to find the node identified by the location, computes the screen location of the node's vertexes, and sends these coordinates back to the technique. BARISTA uses the coordinates to highlight the element, as in Figure 4.4.

The user can then either change the currently selected element through dragging or accept it. At this point, the recorder identifies the node on the accessibility tree as usual (in case the user changed it), checks the node class, and based on this information builds a list

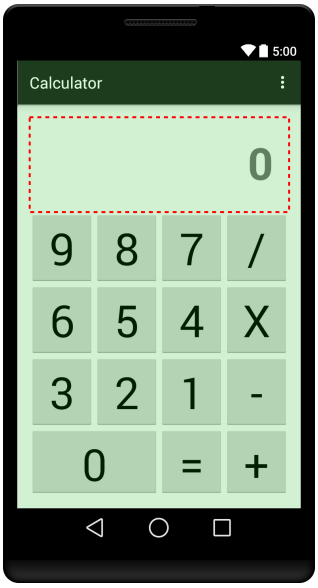


Figure 4.4: Assertion pane in BARISTA.

of assertable properties. The top of the list is populated with properties that are specific to the node. As shown in Figure 4.5, these properties are displayed in the proximity of the selected element. The user can then choose a property and the value to be considered in the assertion, and the technique sends the property and the value to the recorder. The recorder creates an entry in the recorded trace ($ui\text{-}assert\text{-}def$), suitably labels the entry based on the selected assertion property ($ui\text{-}a\text{-}type$), identifies the selector for the assertion ($selector$), and adds the user defined value for the assertion to the properties of the entry ($ui\text{-}a\text{-}props$). After the recorder successfully adds the assertion to its recorded trace, it signals the end of the assertion definition process to BARISTA, which removes the assertion pane from the screen, so that the user can continue to interact with the app.

Activity-Flow-Based Oracles Apps use intents to transfer control flow between app components. The technique allows users to check the properties of intents in their recorded test cases. The user can enable this type of assertions by setting a flag at the beginning of the recording process. The recorder recognizes intents being used within the AUT by reading a log of system messages generated by the Android system. When the recorder detects that the AUT used an intent to transfer control between two app components (by

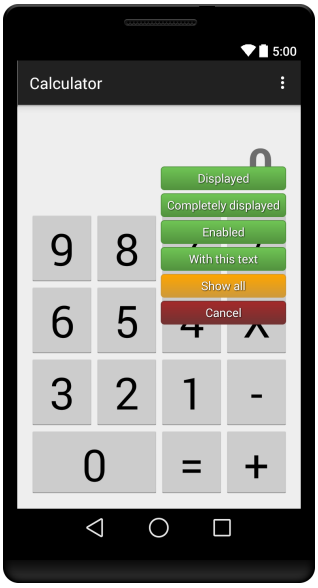


Figure 4.5: Oracle selection in BARISTA.

processing system message entries in the log), it adds an assertion ($af\text{-}assert\text{-}def$) into the recorded trace that checks for the values describing the properties of the intent (action, data, type, category).

4.1.2 Test Case Encoding

The test case encoding phase receives as input the recorded trace and a user-provided flag (*retain-time flag*) that indicates whether the timing of recorded interactions should be preserved. For instance, if a user sets a 30-seconds timer in an alarm clock app and wants to check with an assertion the message displayed when the timer goes off, he or she would set the retain-time flag to true to ensure that the assertion is checked 30 seconds after the timer is started. The test case encoding phase produces as output a test case that faithfully reproduces the actions in the recorded trace. In the current version of the technique, the generated test case is an Android UI test case based on the Espresso framework [49]. In the rest of this section, I illustrate how BARISTA translates the trace into a test case.

The test case encoding phase starts by translating the $main\text{-}activity$ entry of the recorded trace into a statement that loads the starting activity of the recorded interactions. It then

translates actions into statements grouping statements into a test procedure.

Statements that reproduce user interactions and UI-based oracles are divided into three parts. The first part is used by the test case execution engine to retrieve the UI element affected by the action. The technique places the selector ($selector$) of the action in this part of the statement. The second part of the statement consists of the action that the test case execution engine performs on the UI element identified by the first part of the statement. BARISTA encodes this part of the statement with the Espresso API call corresponding to the action being processed ($i\text{-}type$ or $ui\text{-}a\text{-}type$). The third part of the statement accounts for parameters involved in the action and it is action specific. To generate this part of the statement, the technique processes the properties of the action ($i\text{-}props$ or $ui\text{-}a\text{-}props$). Statements representing user-induced system events and activity-flow-based oracles do not follow this structure. Actions representing user-induced system events are translated into statements that call procedures of the currently executing activity. Actions representing activity-flow-based oracles translates into statements that check the properties of intents generated by the execution of the AUT.

The content of the generated test case is affected by the retain-time flag as follows. If the retain-time flag is set, the technique places an additional statement between the statements representing two subsequent actions. This statement pauses the execution of the test cases (but not the execution of the app being tested) for a duration that is equal to the difference of the timestamps associated with the two actions.

Overall, the technique translates interactions and oracles into a single line statement. The one-to-one mapping between actions and statements favors readability and understanding of generated test cases, thus addressing a well-known problem with automatically generated tests.

4.1.3 Test Case Execution

The test case execution phase takes as input the test case produced by the second phase of the technique, together with a user-provided list of devices on which to run the test case, and performs three main tasks: (1) prepare a device environment for the test case execution, (2) execute the test case, and (3) produce the test report.

The first step installs the AUT and the generated test case on all devices in the user-provided list. Once the execution environment is set up, the technique executes the test case on each device in the user-provided list in parallel. The execution of a test case is supported through an extension of the Espresso framework and works as follows. The test case execution engine begins with loading the starting activity of the test. From this point, the engine synchronizes the execution of the test's steps with updates in the UI of the AUT.

The engine processes user interaction and UI-based oracle statements as follows. It first navigates the UI displayed by the device to find the UI element referenced by the action. If the element is not present, the execution of the test case terminates with an error. If the element is present, the execution engine behaves differently according to whether it is processing an interaction or an oracle statement. In the former case, the execution engine injects a motion event into the app or performs an API call on the UI element being targeted by the interaction. In the case of an oracle statement, the execution engine retrieves all elements in the UI that hold the property expressed by the oracle's assertions and checks whether the element targeted by the oracle is one of these elements. If the element is not present, the test case terminates with a failure. Otherwise, the execution continues.

The engine processes user-induced system event statements by mediating the execution of the system event with the Android system. Activity-flow-based oracle statements are processed as follows. During the execution of the test case, the test engine stores intents being sent by the AUT into a buffer. When the engine reaches an oracle statement it checks that the buffer contains an intent with the same properties as the one expressed by the statement. After the execution of such statement, the engine clears the buffer to make sure

that following oracle statements will match new intents.

At the end of the execution, the technique produces a test execution report that contains: (1) the outcome of the test case on each device, (2) the test case execution time, and (3) debug information if an error or failure occurred during execution.

4.2 Evaluation

To assess the expressiveness, efficiency, and ultimately usefulness of BARISTA, I implemented the technique in a prototype tool and performed a user study involving 15 human subjects and 15 real-world Android apps. Because defining oracles is a fundamental part of generating test cases and of the technique, to perform an apple-to-apple comparison I used as a baseline for the evaluation: TESTDROID RECORDER (TR) [84] and ESPRESSO TEST RECORDER (ETR) [50]. The former tool records test cases in the Robotium [181] format while the latter records test cases in the Espresso [49] format. I therefore did not consider pure record and replay tools with no oracle definition capabilities, such as RERAN [46], VALERA [76], and MOSAIC [184]. I considered including ACRT [101] in the study, as it can record tests in Robotium format. Unfortunately, however, ACRT does not work with recent Android versions, so using it would have required us to backport the benchmark applications to an earlier Android version.

In the evaluation, I investigated the following research questions:

- **RQ1:** Can BARISTA record user defined test cases? If so, how does it compare to TR and ETR?
- **RQ2:** Is the test case recording process with BARISTA more efficient than the one with TR and ETR?
- **RQ3:** Does BARISTA's encoding preserve the functionality of the test cases? How does BARISTA compare to TR and ETR in this respect?

Table 4.2: Description of the benchmark apps used to evaluate BARISTA.

ID	*Name*	*Category*	*Installations (K)*	*LOC (K)*
A1	DAILY MONEY	Finance	500 - 1000	10.7
A2	ALARM KLOCK	Tools	500 - 1000	6.1
A3	QUICKDIC	Books	1000 - 5000	289.7
A4	SIMPLE C25K	Health	50 - 100	1.5
A5	COMICS READER	Comics	100 - 500	8.4
A6	CONNECTBOT	Communication	1000 - 5000	24.3
A7	WEATHER NOTIFICATION	Weather	100 - 500	13.2
A8	BARCODE SCANNER	Shopping	100000 - 500000	47.9
A9	MICDROID	Media	1000 - 5000	5.6
A10	EP MOBILE	Medical	50 - 100	31.4
A11	BEECOUNT	Productivity	10 - 50	16.2
A12	BODHI TIMER	Lifestyle	10 - 50	10.5
A13	ANDFHEM	Personalization	10 - 50	60.3
A14	XMP MOD PLAYER	Music & Audio	10 - 50	58.7
A15	WORLD CLOCK	Travel & Local	50 - 100	31.4

- **RQ4:** Can test cases generated by BARISTA run on different devices? How platform independent are they with respect to test cases generated by TR and ETR?

In the remainder of this section, I first describe the benchmarks used in the evaluation. I then present the user study, discuss evaluation results, and conclude illustrating anecdotal evidence of BARISTA's usefulness using feedback from developers that used it.

4.2.1 Benchmarks

For the evaluation of the technique, I used a set of real-world Android apps. Specifically, I used 15 open-source apps from the F-Droid catalog [37] (ETR requires an app source code). The choice of apps is based on three parameters: (1) popularity, (2) diversity, and (3) self-containment. As a popularity measure, I used the number of installations for an app according to the Google Play store [47]. I selected apps from different categories to have a diverse corpus of benchmarks and prioritized apps for which I did not have to build extensive stubs (*e.g.,* apps that do not rely on a hard to-replicate backend database). Table 4.2 shows the lists of apps I used. For each app, the table shows its ID (*ID*), name (*Name*), category (*Category*), the range of its installations (*Installations*), and the lines of code (*LOC*). It is worth noting that none of the apps in Table 4.2 had a reference test suite.

4.2.2 User Study

For the evaluation, I recruited 15 graduate students from three institutions. I asked the participants to perform four tasks: (#1) write natural language test cases (NLTCs), (#2) record NLTCs using TR, (#3) record NLTCs using ETR, and (#4) record NLTCs using BARISTA. Before performing the user study, I conducted a three-hour tools demonstration session to familiarize the participants with the three tools. I did not inform the subjects of which tool was mine and which ones were the baseline (but they obviously could have discovered this by searching the name of the tools).

All participants started from the task #1. In this task I provided the participants with three benchmark apps, so that each app was assigned to three different users. I asked the participants to explore the apps' functionality and then define five NLTCs for each app assigned to them. NLTCs were written purely in natural language, without the use of any framework and without even following any particular structure. After they all completed task #1, I manually analyzed the NLTCs for possible duplicates and checked with the participants in case of ambiguities. Table 4.3 shows the properties of the NLTCs I collected. For each app, the table shows the number of distinct NLTCs (*NLTC*), average number of interactions per test case (*I*), and average number of assertions per test case (*A*). The total number of distinct NLTCs is 215. All NLTCs have at least one assertion. A1 is the app having the NLTC with the highest number of interactions (27), while A11 is the app with the NLTC having the highest number of assertions (10). All NLTCs should pass.

In tasks #2, #3 ,and #4, I asked participants to record NLTCs using TR, ERT, and BARISTA, respectively. For each task, each participant was provided with a set of NLTCs written for three apps. The set of NLTCs provided to a specific participant was different between the three tasks. However, the set of all test cases across the three tasks was the same. I also decided not to give participants NLTCs they wrote, so as to mimic a scenario in which the test specifications are provided by a requirements engineer and the testing is performed by a QA tester. For each of the three tasks, I asked the users to reproduce

Table 4.3: Information on the NLTCs considered: *NLTC* = number of NLTCs for the app; *I* = average number of interactions in NLTCs; *A* = average number of assertions in NLTCs.

ID	*NLTC*	*I*	*A*
A1	15	9.33	3.40
A2	15	7.07	1.40
A3	14	6.21	1.36
A4	14	4.36	3.14
A5	14	3.50	1.93
A6	13	8.92	1.23
A7	14	3.29	2.79
A8	14	2.93	1.86
A9	12	4.08	1.25
A10	15	6.47	3.00
A11	15	6.73	2.20
A12	15	3.67	1.73
A13	15	3.93	3.13
A14	15	4.87	2.47
A15	15	4.47	3.27
Total	215	5.33	2.30

the steps of the NLTCs as faithfully as possible, unless the tool prevented them to do so (*e.g.,* they could skip assertions that the tool was unable to encode). Finally, I grouped participants so that some performed the task #2 before the other two tasks, others started from task #3, and still others started from task #4.

The experimental setup to perform task #2 was structured as follows. I asked users to record NLTCs on a device running Android API level 19. The device was connected to a MacBook Pro (2.3 GHz i7 processor and 8GB memory) running Eclipse 4.4, with TR installed as a plugin. To define an assertion using TR, users might need to specify the Android resource IDs of the element involved in the assertion. I thus made the UIAUTOMATORVIEWER tool [48] available to users, so that they could easily explore an app's UI. In task #3, I asked users to record NLTCs on a device running Android API level 19. The device was connected to a MacBook Pro (2.3 GHz i7 processor and 8GB memory) running Android Studio 2.2 with ETR installed. To perform task #4, I asked users to record NLTCs using a device running API level 19 with BARISTA installed. I did not impose any timeout to perform the three tasks.

Table 4.4: Results of the test case recording process, for each app considered: *C* = number of test cases that could be recorded; *NC* = number of test cases that could not be recorded; *AS* = number of assertions skipped; *AA* = number of assertions altered; and *T* = average recording time in seconds.

ID	TR					ETR					BARISTA				
	C	*NC*	*AS*	*AA*	*T*	*C*	*NC*	*AS*	*AA*	*T*	*C*	*NC*	*AS*	*AA*	*T*
A1	15	0	9	20	176s	15	0	0	36	97s	15	0	2	0	119s
A2	4	11	0	2	108s	15	0	14	3	27s	15	0	0	0	42s
A3	9	5	5	1	11s	13	1	3	9	40s	14	0	2	0	18s
A4	9	5	8	7	27s	14	0	26	13	30s	14	0	3	0	29s
A5	12	2	2	2	38s	14	0	1	20	19s	14	0	0	0	9s
A6	6	7	0	1	18s	13	0	0	13	29s	13	0	0	0	11s
A7	11	3	13	5	14s	13	1	5	21	15s	14	0	0	0	8s
A8	11	3	5	0	25s	14	0	0	17	21s	14	0	0	0	5s
A9	11	1	3	3	23s	12	0	0	12	28s	12	0	0	0	11s
A10	13	2	10	2	61s	14	1	17	8	38s	15	0	0	0	56s
A11	12	3	10	0	66s	15	0	1	13	56s	15	0	0	0	57s
A12	15	0	5	0	25s	15	0	4	14	28s	15	0	0	0	22s
A13	13	2	14	3	123s	15	0	0	39	51s	15	0	0	0	46s
A14	15	0	7	2	97s	11	4	1	24	43s	15	0	2	0	49s
A15	15	0	17	0	83s	14	1	1	35	124s	15	0	2	0	57s
Total	171	44	108	48	60s	208	7	74	277	43s	215	0	11	0	36s

4.2.3 Results

RQ1 : To answer the part of RQ1 about BARISTA's expressiveness, I checked the test cases recorded by users using BARISTA against the corresponding NLTCs. The third part of Table 4.4 (columns below BARISTA header) shows the results of this check. For each app, I report the number of test cases that could be recorded (*C*), the number of test cases that could not be recorded (*NC*), the number of assertions skipped (*AS*), and the number of assertion altered (*AA*). I considered an NLTC as recorded if the generated test case contained all interactions defined in it, and not recorded otherwise. I considered an assertion as skipped if the user did not define it, whereas I considered an assertion as altered if the user defined an assertion with a different meaning from the one in the NLTC. When using BARISTA, participants could record all test cases, skipped 11 assertions (2.2% of the total number of assertions), and did not alter any assertion. The 11 assertions that users could not express with BARISTA do not directly check for properties of UI elements (*e.g.,* an NLTC for A4 states "assert that the alarm rings").

The first (TR) and second (ETR) sections of Table 4.4 help us answer the second part of RQ1, which compares BARISTA to the baseline. 44 test cases could not be recorded using TR. 36 of those could not be recorded because TR altered the functionality of 10 apps, preventing users from performing certain interactions. In the remaining cases, users stopped recording the test case after making a mistake. Even without considering the last eight test cases, which mostly depend on user errors, BARISTA could record 20.1% more test cases than TR. 7 test cases were not recorded using ETR because users stopped recording the test case after making a mistake.

As the table also shows, users skipped 108 assertions while using TR and 74 while using ETR (the assertions skipped while using BARISTA are included in both sets). The reason behind these two high numbers is that TR and ETR offer a limited range of assertable properties. For instance, TR does not allow for checking whether a UI element is clickable or whether an element is checked, while ETR offers only three assertable properties: *text is*, *exisit*, and *does not esist*. In the test cases generated by TR and ETR, I can also note that 48 and 277 assertions (sum of column *AA*) were different from the ones defined in the corresponding NLTCs. An example of such assertion mismatch is an NLTC from A1, for which the user recorded "assert button is enabled" instead of "assert button is clickable". I asked the participants involved why they modified these assertions, and they said that it was because they could not find a way to record the original assertion with the tool. Among the test cases recorded by all tools, BARISTA could faithfully express 65.2% more assertions than TR and $3.8X$ more assertions than ETR.

These results provide initial evidence that BARISTA can record test cases and is more expressive than TR and ETR.

RQ2 : To answer RQ2, I compare the amount of time taken by the participants to record test cases using TR, ETR, and BARISTA. For each app, Table 4.4 reports the average time in seconds (*T(s)* columns) taken to record test cases. The average time is computed

Table 4.5: Results of test case execution: *T* = number of executed test cases; *W* = number of working test cases; *NW* = number of test cases that did not work due to a problem with the tool; and *M* = number of test cases that did not work due to a user mistake.

ID	TR				ETR				BARISTA			
	T	*W*	*NW*	*M*	*T*	*W*	*NW*	*M*	*T*	*W*	*NW*	*M*
A1	15	8	6	1	15	6	9	0	15	15	0	0
A2	4	3	1	0	15	7	8	0	15	15	0	0
A3	9	5	4	0	13	6	7	0	14	14	0	0
A4	9	3	5	1	14	8	6	0	14	12	0	2
A5	12	10	0	2	14	9	5	0	14	13	1	0
A6	6	4	2	0	13	5	8	0	13	13	0	0
A7	11	9	2	0	13	3	6	4	14	11	0	3
A8	11	8	2	1	14	8	6	0	14	14	0	0
A9	11	11	0	0	12	3	8	1	12	12	0	0
A10	13	9	4	0	14	6	9	0	15	15	0	0
A11	12	8	4	0	15	0	15	0	15	15	0	0
A12	15	12	3	0	15	7	7	1	15	15	0	0
A13	13	1	9	3	15	4	11	0	15	15	0	0
A14	15	9	5	1	11	4	7	0	15	15	0	0
A15	15	11	2	2	14	5	9	0	15	15	0	0
Total	171	111	49	11	208	81	121	6	215	209	1	5

considering the test cases that were recorded by all three tools and in which no assertion was skipped. The amount of time associated with each test case is calculated from the moment in which the user recorded the first action to the time in which the user terminated the recording process. Recording test cases with BARISTA was faster than TR for 13 apps and faster then *ETR* for 10 apps. BARISTA has the lowest average recording time considering all apps and it is 32.3% faster than TR and 19.9% faster than ETR.

I can thus conclude that, on average, BARISTA is more efficient in recording test cases than TR and ETR.

RQ3 : To answer the part of RQ3 about BARISTA's correctness, I executed the 215 test cases generated using BARISTA on the device on which they were recorded. I report the execution results in the third part of Table 4.5 (columns below BARISTA header). For each app, I report the number of test cases executed (*T*), the number of test cases that worked correctly (*W*), the number of test cases that terminated with an error or failure due to a problem in the tool encoding or execution phase (*NW*), and the number of test cases that terminated with an error or failure due to a user mistake in the recording process (*M*). I con-

sider a test case as working correctly if it faithfully reproduces the steps in its corresponding NLTC. Across all benchmark apps, 97.2% of the recorded test cases worked correctly, and 12 apps had all test cases working properly. The test case from A5, which is marked as not working, terminated with an error because the file system of the device changed between the time the test case was recorded and the time the test case was executed. The five test cases marked as user mistakes terminated with an assertion failure. In two of these cases, the user asserted the right property but forgot to negate it. In the remaining three test cases, the user asserted the right property but on the wrong UI element. I presented the errors to users and they confirmed their mistakes.

The first and second part of Table 4.5 (columns below TR and ETR headers) lets us answer the second part of RQ3, which compares the correctness of the test cases generated by BARISTA with respect to that of the baseline. Across all benchmark apps, only 64.9% of the recorded test cases with TR worked correctly. This number corresponds to 51.6% of the NLTCs. The 49 test cases classified as not working could not replicate at least one of the interactions from their corresponding NLTCs. Users made 11 mistakes using TR. In the majority of these cases (6), the user entered the wrong resource ID when recording an assertion. In the case of ETR, only 38.9% of the recorded tests worked correctly. 121 test cases did not work because of the following reasons: (1) the UI reference generated by the tool could not identify the corresponding UI element (75 test cases), (3) the tool generated additional actions that changed the behavior of the test case (30 test cases), and (3) the tool did not generate test case actions for certain user interactions (16 test cases). Users made six mistakes using ETR. In all cases, users altered an assertion making the test fail.

Overall, BARISTA nearly doubles the percentages of working test cases compared to TR and ETR. Based on these results, I can answer RQ3 as follows: there is evidence that test cases generated by BARISTA work correctly, and that BARISTA can outperform TR and ETR in this respect.

RQ4 : To answer the part of RQ4 on BARISTA's cross-device compatibility, I executed the test cases recorded using BARISTA on seven (physical) devices: LG G FLEX (D1), MOTOROLA MOTO X (D2), HTC ONE M8 (D3), SONY XPERIA Z3 (D4), SAMSUNG GALAXY S5 (D5), NEXUS 5 (D6), and LG G3 (D7). (Georgia Institute of Technology acquired these devices in early 2015 with the goal of getting a representative set in terms of hardware and vendors.) I executed all the test cases that did not contain a user mistake, and among those, 206 test cases worked on all devices. Overall, the average compatibility rate across all apps and devices was 99.2%. Two test cases (from A13) did not work on D7 because that device adds additional space at the bottom of a `TableLayout` element. The additional space moves the target element of an action out of the screen, preventing BARISTA from successfully interacting with that element. (The two test cases work on D7 by adding a scroll action to the test cases.) Also, one test case (from A13) did not work on D1, D5, and D7 because these devices display an additional element in a `ListView` component. For this reason, an interaction in the test case selects the previous to last element instead of the last element.

To answer the second part of RQ4, which involves comparing cross-device compatibility of test cases generated using BARISTA with respect to the baseline, I executed on the seven devices considered the test cases (that did not contain a user mistake) recorded using TR and ETR. For TR, 108 tests worked on all devices, and the average compatibility rate across all apps and devices was 68.3%. Many of the failing tests also failed on the device on which they were recorded. In addition, TR generated three test cases that did not work on D5: one test (from A9) failed to identify the target element of an interaction based on the x and y coordinates stored in the test case; two tests (from A15) used an index to select the target element of an interaction that was not valid on the device. For ETR, 62 tests worked on all devices, and the average compatibility rate across all apps and devices was 37.3%. Also in this case, many tests failed on the device on which they were recorded as well. In addition ETR generated 2 tests that did not work on D1, 1 test that did not work

on D2, 19 tests that did not work on D4, 3 tests that did not work on D5, and 14 tests that did not work on D7. 37 of these failures were caused by the UI reference generated by the tool. The remaining two failures were caused by an unsatisfiable constraint in the test. Finally, it is worth noting that, whereas for the three BARISTA-generated tests that are not cross-device compatible, the corresponding TR- and ETR-generated tests are also not cross-device compatible, the opposite is not true; that is, for the TR- ETR-and generated tests that are not cross-device compatible, the corresponding BARISTA-generated tests are cross-device compatible.

Based on these results, I can conclude that tests generated using BARISTA can run on different devices in a majority of cases, and that BARISTA generated a greater number of cross-device-compatible tests than TR and ETR.

4.2.4 Developers Feedback

I publicly released BARISTA and could reach to several developers in various companies to introduce the tool and ask them to give me feedback in case they used it. Although this is admittedly anecdotal evidence, I want to report a few excerpts from the feedback I received, which echo some of my claims about BARISTA's usefulness. Some feedback indicates the need for a technique such as BARISTA: *"I have been looking for something like* BARISTA *to help me get into automation for a while"*. Other feedback supports the results of the evaluation on the efficiency of BARISTA: *"Overall, a very interesting tool! For large-scale production apps, this could save us quite some time by generating some of the tests for us"*. Finally, some other feedback points to aspects of the technique that should be improved and that I plan to address in future work: *"There are a few more assertions I'd like to see. For example, testing the number of items in a* `ListView`*"*. I am collecting further feedback and will make it available on the BARISTA's website.

4.2.5 Threats To Validity

There are both internal and external threats to validity associated with the evaluation. In terms of internal validity, the participant of the user study were not familiar with the apps they generated test cases, which may not be the case in real-world situations. However, it is not uncommon for testers to test someone else's software. In terms of external validity, the results might not generalize to other apps. To mitigate this threat, I randomly selected real-world apps. The results might also not generalize to other devices. To mitigate this threat, I selected a representative set of devices in terms of hardware and vendors.

CHAPTER 5

DETECTING CROSS-PLATFORM INCONSISTENCIES

This chapter describes my technique to identify cross-platform inconsistencies for improving software quality before release. The name of the technique is DIFFDROID and it is based on the idea of differential testing. DIFFDROID identifies inconsistencies in the UI of an app when running on different platforms and targets Android apps. I use the term cross-platform inconsistency (CPI) to describe an identified inconsistency. The technique focuses on UI inconsistencies. Because the UI is the component of an app used to define computation inputs and observe computation outputs, an inconsistent behavior in this component can affect the functionality of the app. The rest of this chapter is structured as follows. Section 5.1.2 motivates this work with an empirical study and a motivating example, Section 5.2 details the technique and Section 5.3 discusses the evaluation of DIFFDROID.

5.1 Motivation

5.1.1 Empirical Study

To evaluate the magnitude of the issues in the fragmentation of the Android ecosystem [132], I ran an empirical study with a collaborator [5]. In this study, we investigated fragmentation incompatibilities related to the UI of a mobile platform. To do so, we executed the tests from the `view` package of the Android Compatibility Test Suite (CTS) [64] on devices available at the Amazon Device Farm [4]. The CTS is a test suite that offers a compatibility "mechanism" to device vendors. Specifically, it contains test cases that execute calls to the operating-system API and specify the expected outcome for these calls. The Device Farm is a testing service provided by Amazon that allows to run tests on real mobile devices. We executed tests from the `view` package as they are the ones exercising UI capabilities.

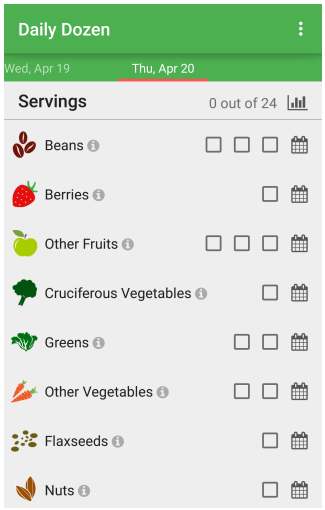


Figure 5.1: DAILY DOZEN app running on LG G3.

We ran the tests on 123 different mobile platforms. These platforms were running 11 different versions of the Android operating system. The number of test executed is 711 per version on average. (The CTS has a different number of tests for each version of the operating system.) In total, the tests revealed 224 failures across 72 unique devices. These failures were caused by 26 unique test cases. These results show that a good percentage of devices exhibit incompatibilities, it is therefore necessary to detect incompatibilities during app testing.

5.1.2 Motivating Example

To further motivate this work I provide an example from a real-world app called DAILY DOZEN [32]—a diet tracking app that has been downloaded more than $50,000$ times and reviewed by more than $1,000$ users. Figure 5.1 shows the `MainAcitivity` of the app running on a LG G3 device, while Figure 5.2 shows the same activity running on a LG Optimus L70 device. Users can use this activity to track their daily food intake by clicking on the checkbox elements.

Figures 5.1 and 5.2 show a CPI for the app. Users can tick the checkbox element associated with the "Cruciferous Vegetables" label on a LG G3 device, but they cannot do

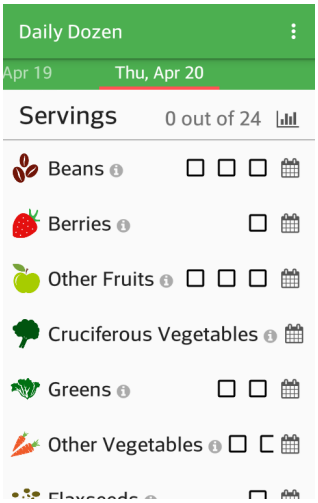


Figure 5.2: DAILY DOZEN app running on LG Optimus L70.

the same on an LG Optimus L70, as that checkbox element is not visible when the app runs on such device. This inconsistency is caused by a bug in the layout file associated with the `MainAcitivity` and is revealed because of the different screen configurations (screen resolution and pixel density) of the two devices.

Bugs of this type can manifests in one of two ways: (1) the checkbox element is present on one device, but not on the other, or (2) the checkbox element is present on both devices, but its visual appearance is different. The checkbox element associated with the "Cruciferous Vegetables" label is an example of the former case. In this case, the difference can be visually perceived, but it can also be identified by comparing the UI hierarchies of the two devices. In fact, the UI hierarchy of the app running on the LG Optimus L70 device does not have a node representing the checkbox element, while such a node is present in the UI hierarchy of the app running on the LG G3 device. The rightmost checkbox element associated with the "Other Vegetables" label is an example of the latter case. In this case, the difference between the two devices can only be perceived visually, as both nodes are present the UI hierarchies of the two devices. These types of issues are far from rare because developers tend to use a limited set of devices (when not only one) during development and testing. In addition, these inconsistencies are hard to detect because this testing process tends to be mostly manual.

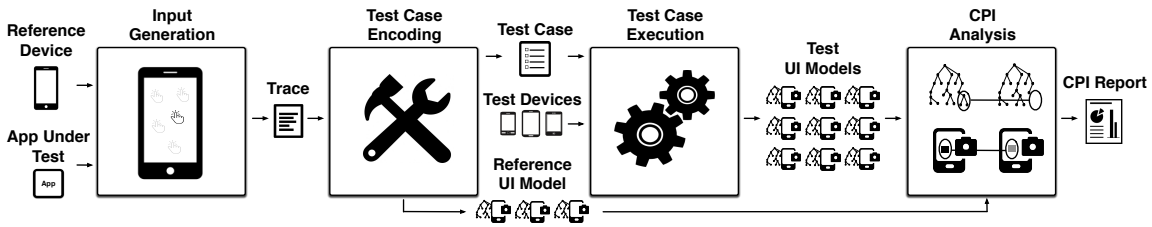


Figure 5.3: High-level overview of DIFFDROID.

Figures 5.1 and 5.2 also highlight the challenges in finding CPIs on mobile devices. Because mobile devices can have different screen configurations, certain differences should not be classified as CPIs. For example, the "Nuts" label is displayed on the LG G3 device, while it is not displayed on the LG Optimus L70 device. This difference should not be considered a CPI: the label is part of a scrollable list, and the former device simply accommodates more list items due to its larger screen.

5.2 Technique

In this section, I present DIFFDROID, my technique for detecting CPIs on mobile devices. The basic idea behind DIFFDROID is to use differential testing to identify such inconsistencies. Figure 5.3 provides a high-level overview of the technique and shows its main phases. Given an app under test (*AUT*) and a reference device, in its *input generation phase*, DIFFDROID dynamically generates inputs with the goal of testing the app's functionality. Before providing the generated inputs to the app, the technique captures the UI state of the app by storing the tree of its UI hierarchy [52] and taking a screenshot of its appearance on the device. The technique logs UI hierarchy trees, screenshots, and generated inputs into the *trace*, which is the input to the following phase: the *test case encoding phase*. In this phase, the technique suitably analyzes the inputs together with UI hierarchy trees to generate a platform-independent test case. While doing so, the technique also creates a *UI model* of the app. The UI model is composed of a list of *window models*; and each window model contains a UI hierarchy tree and corresponding screenshot. I call this UI model the *reference UI model*, as it was generated using the reference device. The *test case execution*

Algorithm 1: Input generation in DIFFDROID.

```
   Input : rd: reference device
           AUT: application under test
           T: input generation timeout
   Output: trace: window models and generated inputs on reference device
1  begin
2      trace = ∅
3      t = GET-CURRENT-TIME()
4      START(rd, AUT)
5      while T < GET-ELAPSED-TIME(t) do
6          root = GET-ROOT(rd)
7          tree = TRAVERSE(root)
8          screenshot = GET-SCREENSHOT(rd)
9          trace.ADD(WINDOW-MODEL(tree, screenshot))
10         input = GENERATE-INPUT(w1, KEY, w2, SYSTEM, w3, TOUCH)
11         INJECT(rd, input)
12         trace.ADD(input)
13     return trace
```

phase takes as input a set of test devices, executes the test case generated by the previous phase on the devices, and produces as output a *UI model* for each device (*test UI models*). Finally, in the *CPI analysis phase*, DIFFDROID performs a differential analysis to compare the reference UI model with the test UI models and generates a report that contains the detected CPIs. The *CPI report* is the output of the technique.

5.2.1 Input Generation

The input generation phase aims to test the functionality of the AUT on a reference device by dynamically generating inputs and providing them to the app. — describe this phase with the help of Algorithm 1. The algorithm takes as inputs the reference device (rd), the AUT (AUT), and a timeout (T), and produces as output a trace ($trace$) that contains window models and generated inputs. I present the abstrax syntax of the trace produced by the algorithm in Figure 5.4.

The algorithm begins with an empty trace (line 2). It then starts the AUT (START) on the reference device (line 4) and subsequently enters its main loop, where it iterates until a timeout is reached.

In the first part of the loop iteration (lines 6-9), the algorithm retrieves the root node of the UI hierarchy (GET-ROOT) and traverses the UI hierarchy (TRAVERSE) to build a tree representation of such hierarchy. Each node in the tree is characterized by the following set of properties: $node\text{-}id$, $node\text{-}type$, $left$, $right$, top, $bottom$, $text$, $checked$, $enabled$,

$$
\begin{array}{lll}
\textit{trace-def} & ::= & \textbf{\textit{trace}}\ \textit{items} \\
\textit{items} & ::= & \textit{window-model-def input-def} \\
 & ::= & \mid \textit{window-model-def input-def items} \\
\textit{window-model-def} & ::= & \textbf{\textit{window-model}}\ \textit{tree-def screenshot-def} \\
\textit{tree-def} & ::= & \textbf{\textit{tree}}\ \textit{root-reference-id}^{*}\ \textit{nodes} \\
\textit{nodes} & ::= & \textit{node-def} \mid \textit{node-def nodes} \\
\textit{node-def} & ::= & \textbf{\textit{node}}\ \textit{reference-id}^{*}\ \textit{node-props children-ids} \\
\textit{node-props} & ::= & \textit{node-id}^{\dagger}\ \textit{node-type}^{\dagger}\ \textit{text}^{\dagger}\ \textit{checkable}^{\ddagger} \\
 & ::= & \textit{clickable}^{\ddagger}\ \textit{focusable}^{\ddagger}\ \textit{scrollable}^{\ddagger} \\
 & ::= & \textit{long-clickable}^{\ddagger}\ \textit{checked}^{\ddagger}\ \textit{enabled}^{\ddagger}\ \textit{focused}^{\ddagger} \\
 & ::= & \textit{selected}^{\ddagger}\ \textit{left}^{*}\ \textit{right}^{*}\ \textit{top}^{*}\ \textit{bottom}^{*} \\
\textit{children-ids} & ::= & \mid \textit{reference-id}^{*}\ \textit{children-ids} \\
\textit{screenshot-def} & ::= & \textbf{\textit{screenshot}}\ \textit{image} \\
\textit{input-def} & ::= & \textbf{\textit{input}}\ \textit{input-type} \\
\textit{input-type} & ::= & \textit{key-input-def} \mid \textit{system-input-def} \\
 & ::= & \mid \textit{touch-input-def} \\
\textit{key-input-def} & ::= & \textbf{\textit{key}}\ \textit{key-value}^{\dagger} \\
\textit{system-input-def} & ::= & \textbf{\textit{system}}\ \textit{system-input-type system-input-props} \\
\textit{system-input-type} & ::= & \textbf{\textit{rotate}} \mid \textbf{\textit{data}} \\
\textit{system-input-props} & ::= & \textit{exprs} \\
\textit{touch-input-def} & ::= & \textbf{\textit{touch}} \mid \textit{coords} \\
\textit{coords} & ::= & \textit{x-coord}^{*}\ \textit{y-coord}^{*}\ \textit{pointer-id}^{*} \\
 & ::= & \mid \textit{x-coord}^{*}\ \textit{y-coord}^{*}\ \textit{pointer-id}^{*}\textit{coords} \\
\textit{exprs} & ::= & \textit{expr} \mid \textit{expr exprs} \\
\textit{expr} & ::= & \textit{boolean} \mid \textit{number} \mid \textit{string}
\end{array}
$$

Figure 5.4: Abstract syntax of the generated trace. "*" indicates that the value is a number, "†" indicates that the value is a string, and "‡" indicates that the value is a boolean.

focused, *selected*, *clickable*, *checkable*, *focusable*, *scrollable*, and *long-clickable*. I selected this set of properties because it is the minimal set of properties that allows the technique to best differentiate nodes in the UI hierarchy (see Section 5.2.4). After building a tree representation of the UI hierarchy, the algorithm captures a screenshot of the AUT (GET-SCREENSHOT). The algorithm then pairs the tree representation of the UI hierarchy with the screenshot of the AUT to define the current window model and adds the model to the trace.

In the second part of the loop iteration (lines 10-12), the algorithm generates an input (GENERATE-INPUT), provides the input to the AUT (INJECT), and adds the input to the trace. DIFFDROID generates three types of inputs (KEY, SYSTEM, and TOUCH) using a weighted random distribution, as done in related work [51]. (It is worth noting that the technique would also work with a different dynamic input generation approach, such as [106, 3, 179, 15, 28, 71, 107, 6].) Key inputs are characterized by the value of the key they are representing; system inputs express a change in the orientation of the device or

Algorithm 2: Test case encoding in DIFFDROID.

```
   Input : trace: window models and generated inputs on reference device
   Output: tc: test case
           RUIM: UI model of the reference device
 1 begin
 2   tc = ∅
 3   RUIM = ∅
 4   foreach item ∈ trace do
 5     if item ≡ WINDOW-MODEL then
 6       newModel = TRUE
 7       foreach wModel ∈ RUIM do
 8         if SAME-TREE(wModel.GET-TREE(), item.GET-TREE())
 9           ∧ SAME-IMAGE(wModel.GET-SCREENSHOT(), item.GET-SCREENSHOT()) then
10           newModel= FALSE
11       if newModel == TRUE then
12         RUIM.ADD(item)
13         stmt = GENERATE-WINDOW-MODEL-STATEMENT(item)
14         tc.ADD(stmt)
15     else
16       stmt = GENERATE-INPUT-STATEMENT(item)
17       tc.ADD(stmt)
18   return tc, RUIM
```

data used to transfer control between components of the AUT; and touch inputs represent clicks or gestures on the device. Note that the technique does not currently remove inputs that do not affect the state of the AUT, but they could be discarded using an approach based on delta debugging [182].

5.2.2 Test Case Encoding

The test case encoding phase aims to generate a platform-independent test case based on the content of the trace created by the input generation phase. I present this phase in Algorithm 2.

The algorithm takes as input the trace ($trace$) generated by the previous phase of the technique, and it produces two outputs: a platform independent test case (tc) and a UI model of the reference decvice ($RUIM$). The algorithm begins with an empty test case (line 2) and an empty UI model (line 3). It then processes the content of the trace in its main loop (lines 4-17).

In the first part of the loop iteration (lines 5-14), the algorithm processes window models. If the currently processed window model has the same tree representation (SAME-TREE) and the same screenshot (SAME-IMAGE) of a window model already added into the reference UI model (lines 7- 10), the model is discarded as superfluous. Function SAME-TREE performs a breadth-first traversal of two trees and compares the value of the

properties of the traversed nodes. If the two trees have different structure, or if their nodes have different properties, the algorithm considers the two trees and corresponding window models to be different. Function SAME-IMAGE compares two screenshots using the complex wavelet structural similarity (CW-SSIM) index [154], which can range between zero (different images) and one (similar images). I motivate the use of this index to compute image similarity in Section 5.2.4. If two screenshots do not have CW-SSIM index equal to one, I consider the corresponding window models to be different. If a window model is not redundant (lines 11-14), the algorithm adds the window model to the reference UI model. It also generates a test case statement (GENERATE-WINDOW-MODEL-STATEMENT) that builds a tree representation of the UI hierarchy and takes a screenshot of the device.

In the second part of the loop iteration (lines 16-17), the algorithm generates a platform-independent statement (GENERATE-INPUT-STATEMENT) that replicates the action of the input in the trace; it then adds the statement to the test case. I define generated statements as being platform-independent because they can run on any device independently from the operating system version and the device configuration (*e.g.,* screen size). The algorithm creates platform-independent statements following the principles presented in in Chapter 4. Platform-independent test cases allow the technique to collect UI models on many different devices, thus increasing the likelihood of identifying CPIs.

5.2.3 Test Case Execution

The test case execution phase aims to collect UI models from a set of test devices. This phase takes as inputs the test case generated by the previous phase of the technique and a set of test devices, and it executes the test case on the set of test devices. The execution is driven by two types of statements: WINDOW-MODEL-STATEMENT and INPUT-STATEMENT. Statements of the former type traverse the UI hierarchy of the AUT to build a tree representation of such hierarchy and capture a screenshot of the AUT. The tree and the screenshot are paired together to form a window model of the test device; this window

Algorithm 3: CPI detection analysis in DIFFDROID.

```
   Input : RUIM: UI model of reference device
           TUIMMap: map of UI models of test devices
   Output: CPIReport: set of cross device inconsistencies
 1 begin
 2   CPIReport = ∅
 3   for i = 0; i < RUIM.length; ++i do
 4     rdModel = RUIM.GET(i)
 5     foreach td ∈ TUIMMap.KEY-SET() do
 6       //node mapping
 7       tdModel = TUIMMap[td].GET(i)
 8       rdTree = rdModel.GET-TREE()
 9       tdTree = tdModel.GET-TREE()
10       nodeMappingMap = ∅
11       foreach rdNode ∈ rdTree do
12         mappedNodeList = ∅
13         foreach tdNode ∈ tdTree do
14           nodeSim = COMPUTE-STRUCTURAL-SIMILARITY(rdTree, rdNode, tdTree, tdNode)
15           if nodeSim ≥ α then
16             mappedNodeList.ADD(MAPPING(nodeSim, tdNode))
17         nodeMappingMap[rdNode] = mappedNodeList
18       FIND-BEST-MAPPING(nodeMappingMap)
19       structural comparison
20       foreach rdNode ∈ rdTree do
21         if nodeMappingMap[rdNode] == ∅ ∧ ¬WITHIN-DYNAMICALLY-SIZED-ELEMENT(rdNode, nodeMappingMap) then
22           CPIReport.ADD(STRUCTURAL-CPI(td, rdNode))
23           nodeMappingMap.REMOVE(rdNode)
24       foreach tdNode ∈ tdTree do
25         mapped = FALSE
26         foreach rdNode ∈ rdTree do
27           if nodeMappingMap[rdNode].CONTAINS(tdNode) then
28             mapped = TRUE
29         if mapped ==FALSE ∧¬WITHIN-DYNAMICALLY-SIZED-ELEMENT(tdNode, nodeMappingMap) then
30           CPIReport.ADD(STRUCTURAL-CPI(td, tdNode))
31       visual comparison
32       rdScreenshot = rdModel.GET-SCREENSHOT()
33       tdScreenshot = tdModel.GET-SCREENSHOT()
34       foreach rdNode ∈ nodeMappingMap.KEY-SET() do
35         tdNode = nodeMappingMap[rdNode].REMOVE(0)
36         rdNodeImage = rdScreenshot.CROP(rdNode)
37         tdNodeImage = tdScreenshot.CROP(tdNode)
38         isInconsistency = COMPUTE-IMAGE-SIMILARITY(rdNodeImage, tdNodeImage)
39         if isInconsistency then
40           CPIReport.ADD(VISUAL-CPI(td, rdNodeImage, tdNodeImage))
41   RANK(CPIReport)
42   return CPIReport
```

model is then added to the UI model of the test device. Statements of the latter type provide an input to the AUT. These inputs are meant to exercise the AUT on the test device in the same way it was exercised in the generation phase.

The output of this phase is a mapping between test devices and corresponding UI models. Generation of UI models for test devices is amenable to parallelization, as the computation of a UI model for one device is completely independent from the computation of the UI model for a different device.

5.2.4 CPI Analysis

The CPI analysis phase aims to identify CPIs in the AUT and is the core of the technique. I present this phase with the help of Algorithm 3. Algorithm 3 takes as inputs the reference UI model ($RUIM$) and the map of UI models of test devices ($TUIMMap$) generated by

the previous phase. The algorithm produces as output a report that lists the identified CPIs ($CPIReport$); this report is also the overall output of DIFFDROID.

The algorithm begins with an empty CPI report (line 2), iterates over each window model ($rdModel$) in the reference UI model (lines 3-40), and compares the window model at hand to the corresponding window model ($tdModel$) in all test UI models (lines 5-40). The comparison between a reference window model and a test window model is divided into two steps. The first step (lines 6-30) matches nodes from the tree representing the UI hierarchy of the reference device ($rdTree$) to nodes from the tree representing the UI hierarchy of the test device ($tdTree$). The second step (lines 31-40) compares the visual representation (image) of matched nodes. The first step can detect *structural CPIs*, which consist of missing or additional nodes. The second step can detect *visual CPIs*, which consist of nodes with different visual representations. Considering the motivating example of Section 5.1.2, the checkbox element associated with the "Cruciferous Vegetables" label is an example of a structural CPI, while the rightmost checkbox element associated with the "Other Vegetables" label is an example of a visual CPI.

The node mapping process begins by initializing the mapping ($nodeMappingMap$) between nodes in the reference tree and nodes in test tree to the empty value (line 10). For each node ($rdNode$) in the reference tree, the algorithm then computes a node similarity value ($nodeSim$) between the reference node and each node ($tdNode$) in the test tree using function COMPUTE-STRUCTURAL-SIMILARITY. This function computes a value between zero and one that represents the structural similarity of two nodes (see Section 5.2.4) If the similarity value is greater or equal than a threshold α, the algorithm stores the similarity value, together with the test node, in a list ($mappedNodeList$). Threshold α is used to avoid matching nodes that are too dissimilar. The choice of the value of α is related to function COMPUTE-STRUCTURAL-SIMILARITY and I describe it in Section 5.2.4. When the algorithm has processed all nodes in the test tree, it stores the mapping between the reference node and the computed list into $nodeMappingMap$ (line 17).

Once $mappedNodeList$ is computed for all reference nodes, the algorithm computes the optimal mapping between reference nodes and test nodes using function FIND-BEST-MAPPING (line 18). For every reference node, this function sorts the elements in the $mappedNodeList$ in descending order based on their node similarity value. The function then finds the $mappedNodeList$ containing the element with the highest node similarity value, maps the reference node associated with the $mappedNodeList$ to the test node associated with the element of the list, and marks the reference node as processed. Finally, the function removes all occurrences of the test node from the $mappedNodeList$ associated with other reference nodes. This process continues until all reference nodes are processed.

After finding the best mapping between nodes in the reference tree and nodes in the test tree (line 18), the algorithm analyzes the mapping to find structural CPIs. The algorithm iterates over each node in the reference tree (lines 19-23) to find reference nodes that do not have a mapping to a node in the test tree. If such a node is found, it means that the node is present in the AUT while running on the reference device, but it is not present in the AUT while running on the test device. A challenging aspect for the classification of the missing nodes is given by the fragmentation of the Android ecosystem. Devices come with different screen configurations, and it could be normal that two devices have a different number of nodes in their UI hierarchies.

Consider again the motivating example of Section 5.1.2, in which the MAINACITVITY displays a list of servings. When the app is running on the LG G3, the device displays eight elements. When the app is running on the LG Optimus L70, conversely, the device displays only seven elements. In this case, the eighth element of the list should not be classified as an inconsistency because the Android system does not represent a node in the UI hierarchy if the node is not visible. For this reason, the algorithm further analyzes the node using function WITHIN-DYNAMICALLY-SIZED-ELEMENT to determine whether or not the node should be reported as inconsistency. If the node does not have an ancestor that is scrollable, the node is reported as an inconsistency. Otherwise, if the node (1) has

a scrollable ancestor and (2) has its preceding or subsequent sibling (depending on the position of the node in the tree) that is matched to a node that is visibile and it is not at the end of the dynamically sized element, then the node is also reported as an inconsistency. In all other cases, the node is not reported as an inconsistency. In case function WITHIN-DYNAMICALLY-SIZED-ELEMENT confirms that the node is an inconsistency, the node is added to the CPI report as a structural CPI. Similarly, the algorithm iterates over nodes in the test tree (lines 24-30) to find test nodes that do not have a mapping to a node in the reference tree. If such a node is found, and the node is not part of a dynamically sized element, the algorithm reports it as a structural CPI.

After these steps, the algorithm visually compares mapped nodes (lines 31-40). This part of the algorithm starts by retrieving (GET-SCREENSHOT) the screenshots of the reference and test devices from their window models. Then, for each node in the reference tree, the algorithm retrieves the test node mapped to it and creates two images ($rdNodeImage$ and $tdNodeImage$) from the two screenshots ($rdScreenshot$ and $tdScreenshot$) using function CROP. At this point, the algorithm compares the two images using function COMPUTE-IMAGE-SIMILARITY, which uses a decision tree classifier to recognize inconsistencies. I describe the decision tree classifier I use in Section 5.2.4. The function uses similar principles as the ones I discussed in the context of function WITHIN-DYNAMICALLY-SIZED-ELEMENT: it does not report as inconsistencies nodes that are partially visible because they are part of a dynamically sized element. If the classifier identifies an inconsistency, the algorithm reports that the reference node and the test node have a visual inconsistency (line 40).

The algorithm then ranks (RANK, in line 41) inconsistencies according to the following principles: (1) structural inconsistencies are ranked at the top, (2) visual inconsistencies that affect all devices with the same characteristics (e.g., version of the Android operating system) are ranked next, and (3) the remaining inconsistencies are listed last. Finally the algorithm returns the CPI report (line 42), which is the output of the technique.

Node Structural Similarity

Function COMPUTE-STRUCTURAL-SIMILARITY in Algorithm 3 computes the structural similarity between two nodes. The inputs to the function are a reference tree ($rdTree$), a reference node ($rdNode$), a test tree ($tdTree$), and a test node ($tdNode$). The output of the function is a value between zero and one ($nodeSim$) that indicates the similarity between the reference node and the test node. This function is necessary, as nodes in the tree are not required to have identifiers, and different versions of the operating system can use different node types to represent the same node.

The function starts by comparing the identifiers of the reference and test nodes. If the identifiers are the same, and they are unique in both reference and test trees, the function sets the node similarity value to one and returns it. In this case, the function sets the similarity value to its highest value because the identifier is a property manually defined by the developer that is meant to uniquely identify nodes in the tree.

If identifiers are not unique, or they are different, the function checks the position of the two nodes in the tree by comparing their XPaths (using the path expression from the root of the tree). If the nodes have the same XPath, the function returns one as their similarity. If the path expressions of the two nodes differ in more than one path component, the function returns zero as similarity value (to avoid matches between nodes that are too distant in the tree). If only one path component is different, which may be due to small differences in the tree representation of the test devices, the function computes the similarity value based on the following properties of the nodes: $checkable$, $clickable$, $focusable$, $scrollable$, $text$, $checked$, $selected$, $long\text{-}clickable$, $enabled$, and $focused$. The similarity value, in this case, is given by the number of matching properties divided by the number of properties. For the evaluation of DIFFDROID I chose 0.9 as the value of α in Algorithm 3 to indicate that do not nodes having more than one property value that differs should not be matched.

Decision Tree Classifier

Function COMPUTE-IMAGE-SIMILARITY in Algorithm 3 uses a decision tree classifier [144] to compute whether the visual representation of two nodes should be reported as a CPI. The decision tree classifier algorithm creates a model that predicts the value of a target variable based on a set of input variables. The algorithm learns the model using a set of training data. In the context, the training data corresponds to images of nodes from the UI hierarchies of apps running on different devices. The training set must also include a set of images exhibiting CPIs. After building the model, function COMPUTE-IMAGE-SIMILARITY follows the set of decisions in the model to predict the target variable. DIFFDROID uses the following variables as inputs to the the classifier:

Complex-Wavelet Structural Similarity Index. The technique uses the Complex-Wavelet Structural Similarity (CW-SSIM) index [154] to compare the structural similarity of the content of two images. CW-SSIM is an image similarity metric robust to small rotations and translations in the images being compared. This characteristic makes the metric especially suitable in the mobile context because different devices have different screen configurations; therefore, the visual representation of two nodes may present minor differences that should not be reported as CPIs.

Earth's Mover Distance of Color Histograms. DIFFDROID uses the Earth's Mover Distance [151] (EMD) of the color histograms of two images to compare the color composition of the images. EMD is a measure of the distance between two distributions and, intuitively, consists of the minimal cost that must be paid to transform one color distribution into the other. I decided to use this metric to take into account the fact that two images may have similar structure but display different colors.

Relative Ratio Change. This technique uses the relative ratio change to assess whether two images differ significantly in their proportions. The relative ratio change is defined

as $RRC = ((w_t/h_t) - (w_r/h_r))/(w_r/h_r)$, where w_t and h_t are the width and height of the test node, while w_r and h_r are the width and height of the reference node. This value allows DIFFDROID to identify nodes whose ratio is altered as a consequence of the placement of other nodes.

Optical Character Recognition Output. DIFFDROID uses the output of optical character recognition [134] (OCR) to assess whether two images display the same text. The classifier takes as input the value of the comparison (equal/not equal). I decided to use the output of OCR because nodes might have the same text in their tree representation but might display the text differently.

The target variable predicted by the classifier indicates whether the visual representation of two nodes should be reported as an inconsistency.

5.3 Evaluation

To determine the practicality and effectiveness of the technique, I implemented the technique in a prototype tool and performed an evaluation of DIFFDROID on a set of real-world apps. The evaluation investigates the following research questions:

- **RQ1:** Can DIFFDROID detect cross-platform inconsistencies in mobile applications while reporting a limited number of false positives?
- **RQ2:** What is the cost of running DIFFDROID?
- **RQ3:** Are there similarities among devices exhibiting CPIs?

5.3.1 Benchmarks and Setup

For the evaluation, I used a set of real-world Android apps. Specifically, I selected five open-source apps from GitHub [43]. I used open-source apps because the testing environment (Espresso) used in the implementation of the technique requires the source code of

Table 5.1: Benchmarks used in the evaluation of DIFFDROID.

ID	*Name*	*Category*	*Version*	*LOC (K)*
A1	BUILDMLEARN	Education	2.5.0	23.6
A2	DAILY DOZEN	Health	10.3	6.3
A3	KITCHEN TIMER	Tools	1.1.6	4.3
A4	OUTLAY	Finance	1.1.3	8
A5	TRANSLATION STUDIO	Books	9.0	51.2

an app to build and run test cases for it. The technique could be directly applied to app executables by changing testing framework.

I selected apps based on three parameters: (1) presence of at least one known UI-based CPI in the app, (2) self-containment, and (3) diversity. In order to find apps containing at least one known CPI, I searched GitHub's tracker system for the following keywords: "android not clickable", "android cut off", and "android missing button". I used these keywords instead of more generic keywords, such as "android compatibility issue", to eliminate results that were not UI issues, which are out of scope for the technique. From the search results, I removed issues that did not correspond to Android apps and issues that I could not reproduce. Finally, I selected apps from different categories to have a diverse corpus of benchmarks, while prioritizing apps for which I did not have to build extensive stubs. Table 5.1 provides a summary description of the apps considered. Columns *Name*, *Category*, *Version*, and *LOC* report the name, category, version, and lines of code for an app.

The analysis performed by DIFFDROID relies on the use of a reference device. I selected an LG G3 running Android *22* as reference device for the evaluation because Georgia Institute of Technology had the device, and it did not exhibit any of the CPIs already known in the benchmarks. To compute the results of Section 5.3.2, I executed the input generation phase of DIFFDROID on the reference device with a timeout of 10 minutes. I chose this value because related work work [147] found that a set of dynamic input generation tools for Android apps hit their maximum coverage within 10 minutes of execution.

DIFFDROID's analysis also requires a set of test devices. I used the AWS Device

Table 5.2: Test devices divided by resolution and version of the operating system.

	Android Version					
Resolution	*19*	*21*	*22*	*23*	*24*	*25*
720 x 1280	18	2	6	2	0	0
768 x 1280	1	0	0	0	0	0
1080 x 1920	33	12	5	6	0	1
1440 x 2560	8	7	8	13	5	1
480 x 800	8	0	0	0	0	0
540 x 960	5	1	3	0	0	0
480 x 854	1	0	1	0	0	0

Farm [4] for this purpose. Table 5.2 reports the number of devices used in the evaluation grouped by resolution (*Resolution*) and version of the operating system (*Android Version*). The versions of the operating system were the ones available to us and supported by the technologies used for the implementation of DIFFDROID. The total number of devices used was 147.

Finally, DIFFDROID uses a decision tree classifier to recognize CPIs. I trained the classifier using the following procedure. First, I selected one device from each category (combination of resolution and Android version) in Table 5.2 and used this set of devices to compute the training set. I then collected CW-SSIM index, EMD value, OCR output, and relative ratio change (inputs to the classifier) for all the nodes in the view hierarchies showing the known UI-based CPIs. (These nodes are not included in the results of the evaluation.) This procedure produced $5,558$ entries on which to train the classifier. I labeled the entries either `true` or `false` based on whether they represented CPIs or not, respectively. I labeled entries by looking at their visual representation and labeled as `true` entries such that, compared to the reference entry, (1) differed in their content structure, (2) differed in terms of color, (3) differed in terms of visibility, (4) visualized a different text, and (5) had a different aspect ratio. Following these guidelines, I labeled 282 entries. I used the Weka data mining framework [170] to generate a C4.5 decision tree classifier. The framework created a classifier of size 33 with 17 leaves in 0.09 seconds. I evaluated the classifier using 10-fold cross validation, resulting in a precision of 0.978 and recall of

Table 5.3: Results of running DIFFDROID. For each benchmark considered: D = number of test devices; *WM)* = number of window models; N_R = number of nodes in UI hierarchy trees of the reference device; N_T = average number of nodes in UI hierarchy trees per test devices; CPI_S = number of structural CPIs; CPI_F = number of functional CPIs; CPI_V = number of CPIs related to changes in the version of the Android system; CPI_C = number of cosmetic CPIs; *FP* = false positives reported by the technique.

ID	*D*	*WM*	N_R	N_T	CPI_S	CPI_F	CPI_V	CPI_C	*FP*
A1	135	19	491	465.2	0	2	7	14	1
A2	138	22	1199	1174.8	2	0	0	22	4
A3	129	13	286	276.6	2	3	0	2	1
A4	125	14	505	481.8	0	1	0	17	2
A5	136	17	486	466	2	3	0	19	8

0.957. The CPI analysis phase was performed on a workstation with 64GB of memory, one Intel Xeon i7-6700K Skylake 4.0GHz processor, running Ubuntu 14.04.

5.3.2 Results

RQ1 : To answer RQ1, I applied the technique to the experimental benchmarks. Table 5.3 reports the results of the evaluation.

The first part of Table 5.3 (columns D, WM, N_R, and N_T) provides a picture of the scale of the analysis. For each benchmark: column D reports the number of test devices used in the test execution phase; column *WM* provides the number of window models generated by the test case encoding phase; N_R is the number of nodes in the UI hierarchy trees for the reference device; and N_T is the average number of nodes in the UI hierarchies for the test devices. The number of devices used for each benchmark differs because, when running the evaluation, certain devices were not available in the AWS Device Farm. For unavailable devices, I attempted to run test cases three times before moving forward. Columns N_R and N_T differ for two reasons: the app might contain a structural CPI, and different devices display a different number of nodes for dynamically sized elements (e.g., list containers). The total number of nodes analyzed across all benchmarks and devices is $387,174$.

The second part of Table 5.3 (columns CPI_S, CPI_F, CPI_V, and CPI_C) presents the CPIs reported by DIFFDROID. I analyzed CPIs reported by the technique and classified them in

four categories: inconsistencies in the UI hierarchy tree that affect the functionality of the app (*structural CPIs*, CPI_S); inconsistencies in the visual representation of a node that affect the functionality of the app (*functional CPIs*, CPI_F); inconsistencies generated by the version of the Android system used to run the benchmark (*version CPIs*, CPI_V);and inconsistencies in the visual representation of a node that do not affect the functionality of the app because the user can infer their meaning given the context in which they are visualized (*cosmetic CPIs*, CPI_C). Structural CPIs correspond to the inconsistencies with the same name I discussed in Section 5.2.4, while functional CPIs, version CPIs, and cosmetic CPIs correspond to the visual CPIs that I also discussed in Section 5.2.4. The results presented in this section are deterministic, as the classification part of the technique is itself deterministic. In addition, I also classified CPIs reported by DIFFDROID that did not correspond to an inconsistency as false positives (*FP*). Finally, I randomly selected 5 nodes in each benchmark on all test devices ($3,315$ total), checked for possible false negatives, and did not find any.

The technique found CPIs in all the benchmarks analyzed: 6 structural CPIs, 9 functional CPIs, 7 version CPIs, and 74 cosmetic CPIs. I now provide an example from each category to better illustrate the identified CPIs and how I classified them.

TRANSLATION STUDIO is a translation app. In the registration form of the app, there is an icon that, when clicked, presents a privacy note to the user. However, on certain devices, the icon is not present, and the user will miss the opportunity to read the privacy note. On these devices, the node of the icon is not present in the UI hierarchy tree of the app, and DIFFDROID reports this difference as an inconsistency (structural CPI).

KITCHEN TIMER offers a timer functionality. The app can be used to start and stop three timers. If one of the timers is started, the label of the timer increases in size, moving the button to stop the timer at the bottom of the screen. On certain devices, the size of the button becomes small enough to prevent users from stopping the timer. In these devices, the node representing the button is present in the UI hiearchy tree, but its visual appearance

differs from that of the corresponding node on the reference device. DIFFDROID reports this difference as an inconsistency (functional CPI).

BUILDMLEARN is an app that assists users in developing Android apps. The app has a menu that can be used to navigate the app. The color of the background of the items in the menu is different when the app is running on devices using Android version 19. In these devices, the color of the background is similar to the color of the text of the menu items, making difficult to read the entries in the menu. DIFFDROID reports this difference as an inconsistency (version CPI).

OUTLAY helps users track their expenses. Users can enter their expenses using a numpad. On certain devices, only roughly one fourth of the numbers is visible. Also in this case, DIFFDROID reports this difference as an inconsistency (cosmetic CPI).

I looked at the nature of the structural, functional, and cosmetic CPIs mentioned above, and discovered tha they can be fixed by changing properties of corresponding elements in the layout files for the apps. I reported the issues found, and their possible solutions, to the developers of the apps involved.

DIFFDROID also reported 16 false positives for the five benchmarks I considered. The false positives reported can be grouped into two categories. The first category (14 false positives) includes nodes that display text with additional spacing at the end. This behavior causes test nodes to have a big relative ratio change, leading the classifier to report them as inconsistencies. To address this issue, I plan to leverage OCR to recognize text boundaries and compute relative ratio changes based on such boundaries. The second category (2 false positives) includes test nodes whose image differed from the reference one in the color distribution, but the difference is such that it cannot be perceived by the human eye. This characteristic resulted in a significantly high EMD value, leading the classifier to report these nodes as inconsistencies. To reduce the number of this kind of false positives, I plan to investigate how the number of bins in the computation of the EMD value affects false positives and performance.

Table 5.4: Cost of running DIFFDROID. T_G = time to encode inputs and compute the reference UI model; T_E = average test execution time per device; T_S = average time to compare UI hierarchies per device; T_{CW} = average CW-SSIM computation time per device; T_{EMD} = average EMD computation time per device; and T_{OCR} = average OCR computation time per device.

ID	T_G	T_E	T_S	T_{CW}	T_{EMD}	T_{OCR}
A1	2080s	474.9s	139ms	443.8s	14.8s	172s
A2	1102s	512.2s	581ms	575.7s	50.8s	586.5s
A3	2772s	329.9s	115ms	246.4s	11.9s	134.5s
A4	851s	651.9s	210ms	275s	15.1s	103.7s
A5	1803s	376s	166ms	217.5s	14.1s	153.5s

Overall, I feel that the current number of false positives generated by DIFFDROID is acceptable. (Moreover, they can be further reduced through improvements of the technique.) I therefore believe that the results presented in this section provide initial evidence that DIFFDROID can detect CPIs in mobile applications while reporting a limited number of false positives.

RQ2 : To answer RQ2, I measured the time taken by each phase of the technique to process the experimental benchmarks. Table 5.4 summarizes the results and reports: the time required to encode dynamically generated inputs as a test case while computing the UI model of the reference device ($T_G(s)$); the average test case execution time per device ($T_E(s)$); the average time required to compare reference UI hierarchies with test UI hierarchies per device ($T_S(ms)$); the average time required to compute CW-SSIM indexes per device ($T_{CW}(s)$); the average time required to compute EMD values per device ($T_{EMD}(s)$); and the average time required to extract text with OCR per device ($T_{OCR}(s)$).

The values in column $T_G(s)$ show that the cost to compute the UI model based on the dynamically generated inputs is not low (but still acceptable), which validates the choice of not performing this task during the input generation phase. The average time to execute test cases is less than the time taken to generate inputs. This happens mainly because test cases are saving significantly less UI hierarchies and screenshots. (The only exception is A4, for which I had to add a 60sec sleep time to make sure the test would go past the

Table 5.5: Devices with a high number of CPIs in the evaluation of DIFFDROID. Column *AV* (Android Version) reports the version of the Android system running on the device.

Device	*Resolution*	*Density*	*AV*
LG Optimus L70	480 x 800	207	19
Samsung Galaxy S3 Mini	480 x 800	233	19
Samsung Galaxy J1 Ace	480 x 800	217	19
Samsung Galaxy J1 Duos	480 x 800	217	19
Samsung Galaxy S Duos	480 x 800	233	19
Samsung Galaxy Grand Neo Plus	480 x 800	187	19
Intex Aqua Y2 Pro	480 x 854	218	19
Samsung Galaxy Light	480 x 800	233	19
Samsung Galaxy Star Advance	480 x 800	217	19
Samsung Galaxy Note 2	720 x 1280	267	19

login screen on the test devices.) In the worst case (A4), test cases took a total of $1,358$ minutes to execute (*D* from Table 5.3 times T_E from Table 5.4). During the evaluation I took advantage of the fact that this task is highly parallelizable and executed test cases on 10 devices at the time, thus reducing the cost roughly by an order of magnitude.

Finally, the last part of Table 5.4 shows that the time required to compare reference UI hierarchies to test UI hierarchies is negligible compared to the time to compute values for the features of the classifier. In the worst case (A2), the CPI analysis phase took $2,791$ minutes to complete (*D* from Table 5.3 times the sum of T_S, T_{CW}, T_{EMD}, and T_{OCR} from Table 5.4). This task is also highly parallelizable. and when running the evaluation I analyzed eight devices at a time. Finally, the most expensive part of the CPI analysis phase consists of the computation of CW-SSIM indexes, which across all apps and all devices took 351.7 seconds on average.

Based on these results, I can conclude that the analysis performed by DIFFDROID can run overnight, at least for the cases considered.

RQ3 : To answer RQ3, in Table 5.5 I ranked devices based on the number of CPIs they exhibited, with the device exhibiting the highest number of CPIs at the top. For each device: column *Device* shows the name of the device; columns *Resolution* and *Density* shows the pixel resolution and density, respectively, of the device; and column *AV* reports the

version of the Android system running on the device. The top nine devices all have low values for resolution and density, and no other test device has these characteristics. This result suggests that developers should consider to include a device with these characteristics when testing their apps. While looking at the relation between inconsistencies and device characteristics, however, I also observed that considering testing devices solely based on resolution and density would have not allowed us to identify all the inconsistencies reported in Table 5.3. In fact, there are inconsistencies that derive from different hardware configurations of the devices (e.g., the presence of a physical menu button). It is also worth noting that, even if all devices in Table 5.5 happen to run Android version 19, I could not find any reason why this version should be particularly problematic.

5.3.3 Threats To Validity

As it is the case for most evaluations, there are both construct and external threats to validity associated with the results. In terms of construct validity, there might be errors in the implementation of the technique. To mitigate this threat, I extensively inspected the results of the evaluation manually. In terms of external validity, the results might not generalize to other apps or CPIs. In particular, I only considered a limited number of apps. This limitation is an artifact of the complexity involved in manually inspecting results deriving from executions on a large set of devices (over 130). To mitigate this threat, I used randomly selected real-world apps from different domains.

CHAPTER 6

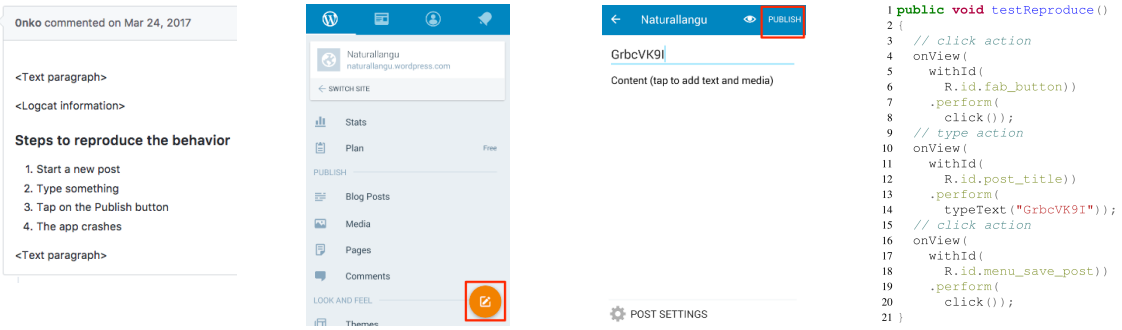
TRANSLATING BUG REPORTS INTO TEST CASES

This chapter details my technique, named YAKUSU, that generates executable test cases from natural language bug reports for improving software quality after release. The technique generates UI tests for mobile applications This task contains three principal challenges. First, extracting actionable information from natural language is a non-trivial task, as it involves interpreting imprecise and context-dependent descriptions. Second, a logical gap can exist between such steps and UI events used in a test case. Third, the sequence of steps may be incomplete. The rest of this chapter is structured as follows. Section 6.1 presents a motivating example, Section 6.2 details the technique, and Section 6.3 discusses the evaluation of YAKUSU.

6.1 Motivating Example

The motivating example is a bug report for WORDPRESS [173], a widely used real-world app for creating web sites and blogs that has been installed over five million times. Figure 6.1a shows the bug report as it appears in WORDPRESS's issue tracking system [1]. The report contains, under the header "Steps to reproduce the behavior", a list of three abstract actions followed by a description of the failure. Figures 6.1b and 6.1c show the screens traversed when performing the actions listed in the report.

As this example shows, actions can be described at different levels of abstraction; some actions refer directly to identifiable UI elements, whereas for other actions there is a logical gap between their description and the corresponding actual UI actions. Abstract action"Tap on the Publish button", for instance, can be easily mapped to the UI action of clicking the button labeled "PUBLISH" in the screen depicted in Figure 6.1c (top right, highlighted). Conversely, the abstract action "Start a new post" corresponds to clicking the round button



(a) Bug report for WORDPRESS app. **(b)** Main screen in WORDPRESS app. **(c)** Edit post screen in WORDPRESS app. **(d)** Test case reproducing bug report for WORDPRESS app.

Figure 6.1: Bug report for WORDPRESS, related screens, and test reproducing the bug.

in the screen shown in Figure 6.1b (bottom right, highlighted). In this case, identifying the corresponding UI action requires a deeper analysis of both the report and the app.

As I describe in detail in Section 6.2, YAKUSU uses a combination of program analysis and natural language processing techniques to identify mappings from abstract actions to UI actions and generate a test case that reproduces the relevant failure. For this example, the test that would be generated by YAKUSU is shown in Figure 6.1d. The test is encoded using the Espresso framework [49], which uses app identifiers to refer to UI elements in the app. Lines 4–8 encode the action "Start a new post" and correspond to clicking the aforementioned round button in Figure 6.1b (element with identifier `fab_button` in the app). Then, lines 10–14 encode the action "Type something" and consist of typing some randomly-generated text on the text box with label "Title" in Figure 6.1c (element with identifier `post_title` in the app). Finally, lines 16–20 encode the action "Tap on the Publish button" by clicking, as described above, the button in the top-right corner of Figure 6.1c (element with identifier `menu_save_post` in the app).

6.2 Technique

This section presents YAKUSU, a technique for translating bug reports written in natural language into test cases. First, YAKUSU combines static analysis and natural language

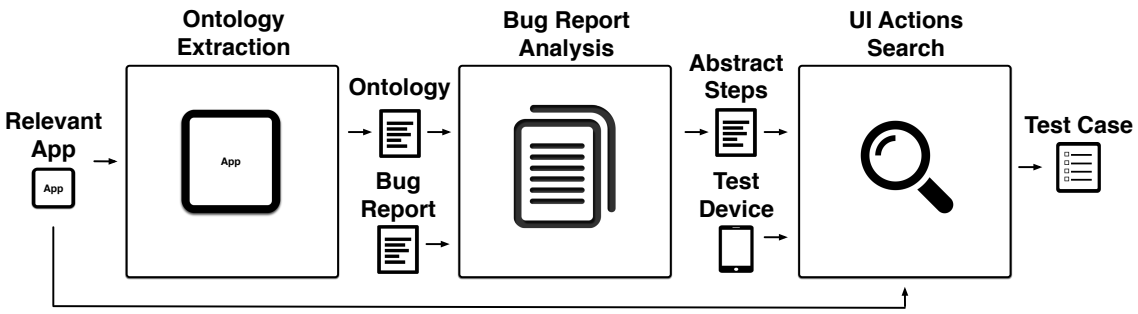


Figure 6.2: High-level overview of YAKUSU.

processing to extract, from a given bug report, a list of abstract steps describing how to reproduce the relevant failure. Then, YAKUSU performs a dynamic search, guided by the previously extracted list of abstract steps, to find a set of UI actions that match these steps.

Figure 6.2 provides an overview of YAKUSU's workflow, which consists of three phases. The *ontology extraction* phase takes the relevant app as input and produces as output the app ontology, which describes the elements available in the UI. The *bug report analysis* phase takes as input the ontology and the bug report and produces as output a list of abstract steps that the execution needs to follow to reproduce the failure documented in the report. Conceptually, the ontology allows for binding the vocabulary used in the bug report with the elements in the UI. This binding is important for producing the list of abstract steps. Finally, the *UI actions search* phase takes as input the list of abstract steps and produces a concrete test case that reproduces the relevant failure. In this phase, the technique runs the relevant app on a test device and tries to map the list of abstract steps provided as input into UI actions. When all abstract steps are mapped, YAKUSU encodes the identified UI actions as a test case that can be run on the app, which is the final output of the technique. The rest of this section describes each phase in detail.

6.2.1 Ontology Extraction

This phase extracts a machine-comprehensible description of the UI elements; this description supports the construction of the mapping between the vocabulary used in the bug

report and the UI elements in the relevant app, which takes place in the next phase (see Section 6.2.2). To create this description, YAKUSU statically analyzes the app using a two-pronged approach: first, it analyzes the UI resource files to identify the widgets that are created statically; then, it analyzes the source code of the app to identify additional widgets that are created at runtime. For this latter analysis, YAKUSU builds a graph of the app that models the creation of UI elements and the modification of their state through setters. Nodes in the graph represent creation and modification operations. Edges represent def-use relations between such operations. Using this graph, YAKUSU can therefore identify (1) which UI elements are dynamically created and (2) what their properties are. For instance, YAKUSU would be able to identify the creation of a button, its label, the callbacks associated with the button, and so on.

YAKUSU stores the identified UI elements, together with (some of) their properties, as tuples. The set of these tuples constitutes the ontology for the app. Specifically, YAKUSU stores in the ontology three kinds of properties for a given UI element: (1) its *label*, (2) the name of the file that contains its associated *icon*, and (3) its *identifier*. (If one or more of these properties is not present, YAKUSU simply stores an empty value for it.)

I selected these three properties because they are particularly suitable for characterizing and identifying a UI element, as I now illustrate. For an element that can display a label, users tend to use such label to refer to this element in bug reports. As an example, consider the step "Tap on the Publish button" in the example of Section 6.1, which uses the label "Publish" of the button to refer to it. Similarly, for a UI element represented by an icon, users often use the name of the object represented by the icon to refer to that element. In this case, there is no textual property to store, so YAKUSU stores in the tuple the name of the file that contains the icon, under the assumption that such name is representative of the icon. As an example, consider the step "Press on attach", from one of the bug reports I used in the evaluation (Section 6.3). In this case, "attach" refers to a paper clip icon whose corresponding filename is `attachFileImage`. Finally, users may refer to an element

of the UI based on its functionality, which may not be reflected in the visual aspect of the element. Using the identifier of an element in the app allows YAKUSU to handle some of these cases, as developers often define identifiers based on the functionality of their corresponding element. The step "Select a Client", for instance, is present in another bug report from the evaluation and is used to refer to one of the elements in a list of clients. Because the identifier for a client is `tv_clientName`, YAKUSU is able to match that step with the correct element using its identifier.

It is worth noting that labels are stored unchanged, whereas YAKUSU performs some normalization for properties icon and identifier to facilitate the analysis in the following phases of the technique. In particular, the technique replaces underscores with spaces and splits apart composite words that follow a camel case convention, both of which are common occurrences in apps [133].

6.2.2 Bug Report Analysis

This phase aims to extract from a bug report the sequence of abstract steps to be performed on the UI of the relevant app for reproducing the relevant failure described in the report. Because bug reports are typically written in natural language, YAKUSU analyzes their content using natural language processing (NLP) techniques, translating the text into dependency trees [83, 110]. (A dependency tree is a directed graph that captures the syntactic structure of a sentence and provides a representation of grammatical relations between words in the sentence.) The tree is characterized by a root word and by relations that connect pair of words in the sentence. Two words involved in a relation are also defined as head and dependent, with the direction of the relation going from the head to the dependent. The technique uses dependency trees based on the Universal Dependency schema [34]. In this schema, frequently used relations can be broken into two sets: *clausal relations*, which describe syntactic roles with respect to a predicate (often a verb), and *modifier relations*, which categorize how dependents can modify their heads. Figure 6.3 provides an example

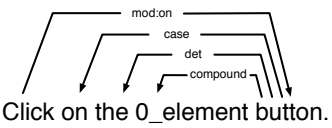


Figure 6.3: Dependency tree computed by YAKUSU.

of a dependency tree. In the remainder of this section, I provide more details on how the dependency tree is computed and processed with the help of Algorithm 4.

The algorithm takes as input the ontology of the relevant app ($ontology$) and the text of the bug report (br), and produces as output a list of abstract steps ($aSteps$) that represent the list of steps described in the report. An abstract step is a tuple $\langle action, target, props \rangle$, where $action$ is a word describing the UI action to be performed, $target$ is a list of words describing the UI element to be exercised by $action$, and $props$ is the list of properties affecting the behavior of $action$. The algorithm begins with an empty list of abstract steps (line 2) and starts by analyzing the bug report (function GET-A-STEPS-TEXT) to identify the portion of text that describes how to reproduce the relevant failure. GET-A-STEPS-TEXT uses a set of heuristics defined based on guidelines on how to report bugs for mobile apps [92, 61, 44, 18]. More precisely, the technique first checks whether the bug report has a section whose content or heading contains the lemma "reproduce". If so, it considers the identified section as the one describing how to reproduce the relevant failure (as it is common practice to use this lemma to describe such section). If such a section is not present, YAKUSU tries to identify the text describing abstract steps by checking whether the report contains a list of bullet points. If neither the section nor the bullet points are present, the technique considers the complete text of the report as relevant. For the bug report of Figure 6.1a, the technique would identify the section with header "Steps to reproduce the behavior", which contains lemma "reproduce", as the text describing the sequence of abstract steps needed to reproduce the failure.

After identifying the right portion of text, YAKUSU preprocesses such text (function PREPROCESS-TEXT) to simplify the subsequent analysis. PREPROCESS-TEXT performs

Algorithm 4: Bug report analysis in YAKUSU.

```
   Input : ontology: ontology of the app
           br: text of the bug report
   Output: aSteps: list of abstract steps describing how to reproduce the relevant failure
 1 begin
 2     aSteps = []
 3     refMap = ∅
 4     text = GET-A-STEPS-TEXT(br)
 5     text = PREPROCESS-TEXT(text, ontology, refMap)
 6     foreach sentence ∈ text do
 7         foreach clause ∈ sentence do
 8             tree = GET-DEPENDENCY-TREE(clause)
 9             root = GET-ROOT(tree)
10             if root ∈ {click, type, scroll, swipe, rotate, ...} then
11                 action = root
12                 target = EXTRACT-TARGET(action, tree, refMap)
13                 props = EXTRACT-PROPS(action, target, tree, refMap)
14                 aStep = CREATE-A-STEP(action, target, props)
15                 aSteps.ADD(aStep)
16             else
17                 if SEMANTICALLY-RELATED(clause, ontology, refMap) then
18                     gAStep = CREATE-GENERIC-A-STEP(null, clause, [])
19                     aSteps.ADD(gAStep)
20     return aSteps
```

three standard NLP operations: noise removal, lexicon normalization, and object standardization [122]. The technique, however, specializes these operations to the domain of bug reports for mobile apps. First, for *noise removal*, YAKUSU discards content within parenthesis, which is in the experience unnecessary for reproduction. For instance, the sentence "*(don't add anything but just)* Press menu button and Reconnect" (from one of the benchmarks in Section 6.3) is changed to "Press menu button and Reconnect". Second, for *lexicon normalization*, YAKUSU normalizes non-standard words to their canonical form to simplify parsing and understanding of actions. (I identified such words from a set of more than 400 tutorials on mobile apps that I collected on the web, by manually inspecting the tutorials for words that are not app specific and refer to actions on the UI.) As an example, in the bug report of Figure 6.1a, YAKUSU changes "*Tap* on the Publish button" to "*Click* on the Publish button". Finally, for *object standardization*, the technique simplifies the text that refers to the target of an action by leveraging the fact that the text displayed by an app usually follows a title or sentence-case convention [153]. Specifically, YAKUSU tries to identify sequences of words in title or sentence-case format that are also textual properties in the ontology of the app and (1) replaces such sequences with a freshly created textual reference ID and (2) adds to $refMap$ the mapping between the ID and the original text in the ontology. For the bug report from Section 6.1, YAKUSU transforms the sentence "Click

on the *Publish* button" into "Click on the *0_element* button" and associates "0_element" to text "`Publish`" in $refMap$. This step is particularly useful when the text being replaced contains multiple words, as it simplifies the later analysis of dependency trees.

After preprocessing the text of the bug report, YAKUSU enters its main loop (lines 6–19), where it analyzes each sentence in the text. More specifically, it analyzes each clause that appears in a sentence, as each of them can specify a different action on the UI. For example, the sentence "Press menu button and Reconnect" contains two clauses connected by a coordinating conjunction, and each clause specifies a different action: (1) open the menu of the app and (2) perform the "Reconnect" action. To identify clauses from sentences, the technique leverages related work [10] that parses a dependency parse tree recursively and, at each step, predicts whether an edge should yield an independent clause.

For each clause, the algorithm computes the clause's dependency tree (GET-DEPENDENCY-TREE). Figure 6.3 provides, as an example, the dependency tree for clause "Click on the 0_element button". The algorithm analyzes the root word of the tree (*e.g.,* "Click" in Figure 6.3) to assess whether it refers to a UI action using, as I discussed earlier, the domain knowledge I distilled from 400 tutorials (Line 10 in Algorithm 4). If the root word of the dependency tree refers to a UI action, YAKUSU further analyzes the tree to extract the properties of the action and the action target, which is the element in the UI affected by the action and, for certain actions, can be missing. Similarity, the list of properties of an action can be empty. For example, the clause "Type something" from the example of Section 6.1 does not specify the target of the action. In this case, the technique would consider any editable UI element as a possible target for the action.

When analyzing a dependency tree, YAKUSU looks for two types of information: the list of words representing the target (EXTRACT-TARGET) and the list of properties affecting the behavior of the action (EXTRACT-PROPS). To identify words that are affected by the action (*i.e.,* the target of the action), the technique analyzes the subtrees rooted at (1) core dependent relations and (2) non-core dependent relations that are associated with the

root word (the action) of the dependency tree. In the case of the dependency tree of Figure 6.3, the technique would identify as the target of the action the subtree rooted at the word associated with the nominal modifier relation (`nmod:on`). The subtree includes the words "on the 0_element button". YAKUSU further analyzes the subtree and, if it contains a textual reference (identified using $refMap$), it uses the text associated with the reference as the target of the action ("Publish", in the case of the example). Otherwise, it uses as target all the words in the subtree. To identify words that detail the behavior of an action, the technique analyzes modifier relations, core dependent relations, and non-core dependent relations [34] associated with the root word (the action) of the dependency tree. For example, in the clause "long click retweet icon underneath tweet" (from one of the benchmarks in Section 6.3), the action "click" is affected by the word "long" through an adverbial modifier relation. YAKUSU identifies such relation, captures the precise action (`long click`) to be performed on the UI (line 11), and stores this information in the list of properties of the action. After the technique determines the action ($action$), the target ($target$), and the properties ($props$) from the clause, it encodes this information into an abstract step (CREATE-A-STEP) and stores the step in the abstract steps list (line 15). I call the encoding of action, target, and properties an *abstract step* because the following phase of the technique will try to find how to concretely execute the step on the relevant app (*i.e.,* find the corresponding concrete UI action).

When there is a logical gap between the description contained in the clause and the corresponding actions to be performed on the UI (*i.e.,* the root word is not $click$, $type$, $scroll$, and so on), the technique assesses whether the clause relates to an element of the UI by semantically comparing (SEMANTICALLY-RELATED) the content of the clause with the elements in the ontology of the relevant app. If the clause is semantically related to an element of the UI, YAKUSU treats the clause as the target of an abstract step as in some cases users describe actions in terms of the elements of the UI. The clause "Start a new post" (from the motivating example in Section 6.1) is an example of such situation. More

precisely, YAKUSU compares the content of the clause with the components (text, icon, identifier) of the tuples in the ontology using word embeddings computed from a word2vec model [119, 120], which offers a mathematical representation of the meaning of a word. Specifically, word2vec produces a vector space from a large corpus of text, where each word in the corpus is assigned to a vector in the space. Vectors are positioned in the space such that words that share common contexts are located in close proximity to one another. The technique computes the word2vec model from a corpus of 100 billion words [60], as word vectorization requires training on very large sets of words [124]. YAKUSU represents the clause as a vector computed by averaging the vectors of the words in the clause, after removing stopwords as they introduce unnecessary noise. By taking the average, the technique is able to incorporate the meaning of every word in the vector representation of the clause. YAKUSU computes the vector of the components in the tuples of the ontology in the same way. It then compares the vector of the clause with the vector of each component by computing the cosine similarity of the two vectors. The similarity value ranges between $[-1.0, 1.0]$, where 1.0 corresponds to the highest similarity. The technique considers a clause to be semantically related to the a UI element if their cosine similarity value is greater than 0.5. I computed this value empirically using a set of bug reports considered for training purposes; this set of bug reports was not used in the evaluation of Section 6.3.

For example, YAKUSU associates the clause "Start a new post" (from the motivating example of Section 6.1) to the text component "`New post`", as the similarity value with this element of the UI is 0.87. When the technique finds a clause to be semantically related to an element in the ontology, it creates an abstract step (CREATE-GENERIC-A-STEP) and adds the step to the list of abstract steps (line 19). Generic abstract steps differ from the abstract steps created by function CREATE-A-STEP; for these steps, the action to be performed will be determined dynamically, during the next phase of the technique, by introspecting the runtime properties of the UI element identified. Finally, the algorithm terminates by returning the list of abstract steps (line 20).

6.2.3 UI Actions Search

This phase takes as input the abstract steps produced in the previous phase and a device on which to run the app and produces as output a concrete test that reproduces the relevant failure. It dynamically explores the relevant app looking for a sequence of UI actions that match the input abstract steps. The technique generates test cases dynamically, rather than statically, because I found that the app navigation models computed by state-of-the-art static analysis tools are too imprecise and incomplete to be used in this context.

Algorithm 5 describes how YAKUSU searches for test cases. The algorithm takes as input the list of abstract steps ($aSteps$), the relevant app (ra), and a test device (td). The output of the algorithm is a test case (tc), which is also the final output of the technique. At a high level, the algorithm explores the relevant app to find a mapping between abstract steps and UI actions to perform on specific elements of the UI. The state of the search is represented as a triple containing (1) the list of abstract steps not yet successfully processed, (2) the list of abstract steps already processed, and (3) the list of UI actions corresponding to the abstract steps processed so far. Lines 2 and 3 initialize the state. At line 2, the algorithm initializes tc with an empty list and $states$ and $pStates$ with an empty set. $states$ denotes the set of states yet to be explored, while $pStates$ denotes the set of states already explored. Line 3 assigns the initial search state $\langle aSteps, [], [] \rangle$ to set $states$. In the initial state, the list of remaining abstract steps corresponds to the list of steps provided on input, whereas the list of processed steps and its corresponding list of UI actions are empty.

After these initialization steps, the algorithm starts to execute its main loop, where each loop iteration processes one state (lines 4–69). The first step in the main loop selects the most promising state for the search (line 5) by calling FIND-BEST-STATE. This function selects the state with the highest number of successfully processed abstract actions, choosing one state randomly in case of ties, returns it, and removes it from set $states$. This depth-first search is often faster for mobile apps, given the cost for restarting the app in alternative search strategies [28]. After selecting the state to be processed, the algorithm starts the rel-

Algorithm 5: UI actions search in YAKUSU.

```
   Input : aSteps: list of abstract steps describing how to reproduce the relevant failure
           ra: relevant app
           td: test device
   Output: tc: test case that reproduces the bug report
 1 begin
 2   tc = [], states = ∅, pStates = ∅
 3   states.ADD(STATE(aSteps, [], []))
 4   while states ≠ ∅ do
 5     state = FIND-BEST-STATE(states)
 6     START(td, ra)
 7     RESTORE(td, ra, state)
 8     rASteps = state.GET-REMAINING-A-STEPS()
 9     while rASteps ≠ [] do
10       aStep = rASteps.REMOVE(0)
11       //Case 1: Abstract steps without a UI element bound to them
12       if aStep == ASTEP ∧ ¬aStep.HAS-ELEMENT() then
13         action = aStep.GET-ACTION()
14         target = aStep.GET-TARGET()
15         ps = aStep.GET-PROPS()
16         elements = FIND-UI-ELEMENT(td,ra,action,target,ps)
17         if elements ≠ [] then
18           element =elements.REMOVE(0)
19           foreach aElement ∈ elements do
20             cstate = state.COPY()
21             caStep = aStep.COPY()
22             crASteps = rASteps.COPY()
23             caStep.SET-ELEMENT(aElement)
24             crASteps.ADD(0, caStep)
25             cstate.SET-REMAINING-A-STEPS(crASteps)
26             states.ADD(cstate)
27           aStep.SET-ELEMENT(element)
28           rASteps.ADD(0, aStep)
29           continue
30         else
31           rASteps.ADD(0, aStep)
32           if aStep.GET-RANDOM-COUNT() < α then
33             rUIAction = RANDOM-UI-ACTION(td, ra)
34             state.ADD-UI-ACTION(rUIAction)
35             PERFORM(td, ra, rUIAction)
36             continue
37           else
38             break
39       //Case 2: Abstract steps with a UI element bound to them
40       else if aStep == ASTEP ∧ aStep.HAS-ELEMENT() then
41         element = aStep.GET-ELEMENT()
42         if FROM-HEURISTIC(element) then
43           hUIAction = HEURISTIC-UI-ACTION(element)
44           state.ADD-UI-ACTION(hUIAction)
45           PERFORM(td, ra, hUIAction)
46         action = aStep.GET-ACTION()
47         if action == null then
48           action = FIND-ACTION(td, ra, element)
49         ps = aStep.GET-PROPS()
50         nUIAction = UI-ACTION(action, element, ps)
51         state.ADD-UI-ACTION(nUIAction)
52         state.GET-PROCESSED-A-STEPS().ADD(aStep)
53         PERFORM(td, ra, nUIAction)
54         continue
55       //Case 3: Generic abstract steps
56       else if aStep == GENERICASTEP then
57         sstate = state.COPY()
58         srASteps = rASteps.COPY()
59         sstate.SET-REMAINING-A-STEPS(srASteps)
60         states.ADD(sstate)
61         target = aStep.GET-TARGET()
62         nAStep =CREATE-A-STEP(null, target, [])
63         rASteps.ADD(0, nAStep)
64         continue
65     if rASteps == [] then
66       tc = GENERATE-TEST-CASE(state.GET-UI-ACTIONS())
67       return tc
68     else
69       pStates.ADD(state)
70   state = FIND-BEST-STATE(pStates)
71   tc = GENERATE-TEST-CASE(state.GET-UI-ACTIONS())
72   return tc
```

evant app on the test device (START) and restores the state of the app by running the list of UI actions associated with the list of already-processed abstract steps (RESTORE). It then calls function GET-REMAINING-A-STEPS to extract the list of remaining abstract steps to

process on the given state ($rASteps$). The inner loop (lines 9–64) processes these abstract steps. In the following, I refer to an abstract step whose action (*e.g.,* click, rotate, scroll) has not been determined as a *generic abstract step*. For example, the first step in the bug report from Figure 6.1a, "Start a new post", is a case of generic abstract step. Each iteration of the inner loop handles one of the following three types of abstract steps: abstract steps without a UI element bound to them, abstract steps with a UI element bound to them, and generic abstract steps.

Case 1: Abstract steps without a UI element bound to them In this first case (lines 12–38), the algorithm must first find a UI element that matches the target specified by the abstract step and then perform the corresponding action on it. To do so, it first extracts the action (GET-ACTION), the target (GET-TARGET), and the properties (GET-PROPS) from the abstract step. It then looks for a potentially matching UI element (FIND-UI-ELEMENT) by processing the properties of elements currently visible in the UI of the relevant app. Specifically, function FIND-UI-ELEMENT compares the textual content of the target with the properties of UI visible elements using word embeddings computed, also in this case, from a word2vec model [119, 120]. The comparison is based on the same types of UI element properties extracted to compute the ontology, namely, label, icon, and identifier. A UI element is considered a match for the target if the cosine similarity between one of the element's properties vector representation and the target's vector representation is above 0.5. The computation of vector representations and their comparison follows the same approach described in Section 6.2.2. In the example from Section 6.1, the technique would find that the button with label "PUBLISH", appearing at the top-right corner in Figure 6.1c, is a match for target "Publish" of the abstract step generated for the sentence "Tap on the Publish button", as their vector representations have cosine similarity 1.0 (they are the same word). Function FIND-UI-ELEMENT uses two heuristics to also "reveal" elements within the screen of the relevant app that may not be readily visible. These heuristics try to identify

this hidden UI elements by opening the menus and scrolling through the lists, respectively, in the current screen of the app.

At this point (line 16), $elements$ stores (in descending order of cosine similarity value) the set of potential UI element candidates returned by FIND-UI-ELEMENT. If this set is empty (lines 30–38), the search for candidates was unsuccessful, which typically happens when a step was missing in the bug report. In that case, the algorithm generates a random UI action ($rUIAction$) and continues to the next iteration, trying to fill the gap in the report, unless the number of random UI actions generated for the current abstract step exceeded a predefined threshold (α). If the search is successful (lines 17–29), set $elements$ is non empty. YAKUSU extracts from this set the element with highest cosine similarity, assigns the element to the abstract step, and reprocesses the abstract step with this UI element bound to it as discussed in the next paragraph (Case 2). For the other elements in the set, the algorithm conceptually forks the execution (lines 19–26) by copying the search state and setting the top remaining abstract step ($aStep$) to have its UI element ($aElement$) adjusted accordingly. These copied states will be processed if the technique does not successfully process all abstract steps in the current execution.

Case 2: Abstract steps with a UI element bound to them In this second case, the algorithm performs a UI action on the element bound to the abstract step (lines 40–54). To do so, the technique either extracts the action associated with the abstract step, or identifies one if there is no associated action (which is possible after a generic abstract step has been processed, as discussed in Case 3). At this point, the technique saves the action in the form of a UI action in the state, adds the abstract step to the list of satisfied steps associated with the state, executes the UI action on the relevant app, and moves forward to analyze the next abstract steps that needs to be processed (line 54).

Case 3: Generic abstract steps In this third and last case (lines 56–64), the algorithm processes steps that may be related to some UI element in the relevant app, but do not

specify any action to be performed on such element. These steps typically either correspond to vaguely expressed actions or are not actual steps. The algorithm accounts for these cases by virtually forking the execution of the relevant app: one execution tries to perform the generic abstract step; the other simply skips this step. To do so, YAKUSU adds to the list of current states a copy of the state ($sstate$) in which the generic abstract step is removed from the list of steps to be processed (line 60). Then, YAKUSU continues the exploration for the current state by generating an abstract step whose target is the one associated with the generic abstract step and whose action is undefined. Then, the technique adds the abstract step to the beginning of the list of steps to be processed and moves forward to process the generic abstract step as an abstract step (line 64).

When the technique finishes processing the abstract steps in a state, it checks the content of the list of remaining abstract steps (lines 65–69). If the list is empty, the search process successfully terminates returning a test case with the list of UI actions associated with the current state (GENERATE-TEST-CASE). In case the list is not empty, YAKUSU adds the state to the list of processed states (line 69) and continues. When the technique is not able to successfully process all abstract steps in any of the states analyzed (line 70), it finds the state that satisfied the highest number of abstract steps (FIND-BEST-STATE) and generates a "partial" test case that consists of the UI actions associated with that state.

6.3 Evaluation

This section discusses the evaluation of the technique. To assess the expressiveness and efficiency of YAKUSU, we investigated the following research questions:

- **RQ1:** Can YAKUSU translate bug reports written in natural language into executable test cases?
- **RQ2:** What is the cost of running YAKUSU?

6.3.1 Experimental Benchmarks and Setup

I used a set of bug reports from real-world apps to evaluate YAKUSU. As the use of Espresso to encode test cases requires the source code of the app to be available, I focused on apps from GitHub [43]. I queried the GitHub database for issues using keywords "android", "crash", "reproduce", and "version" and considered only issues submitted after January 1st, 2017. I used keyword "android" to find issues that relate to Android apps; I included keyword "crash" to find issues that could be easily verified; I used keyword "reproduce" because I was interested in bug reports that describe how to replicate a bug; finally, I included keyword "version" to make sure that the specific version of the app and operating system involved in the issue were available. I considered only issues created after January 1st 2017 to avoid considering apps that could have outdated dependencies.

This search returned 2709 issues, of which I randomly selected 100 for further processing. As a side note, I found that 79 of these 100 issues had been created by individuals who did not perform any commit to the repository, indicating that they were users not involved in the development of the corresponding app.

To be able to answer RQ1 in an accurate way, I first had to make sure that the issues considered were indeed reproducible. To that end, for each of the 100 issues selected, I first tried to build the version of the app specified in the report. In case a version was missing, I used the the most recent working commit before the date of the report to build the app. Somehow surprisingly, I found that building and setting up apps can be fairly complex and time consuming. In some cases, for example, I had to update outdated dependencies or set up server components for an app to compile and run. Overall, I could build and setup 91 of the 100 apps associated with the set of bug reports considered. For the remaining 9 apps, I either encountered compilation errors that I could not fix (7 apps) or could not find the code related t the issue because the repository was initially used only for bug reporting (2 apps). I then tried to manually reproduce the issues reported for the 91 apps that I was able to build and was successful for 62 of them. There were several reasons why

I could not reproduce the issues involving the remaining 29 apps. In 9 cases, the report contained only a stack trace, and this trace did not provide us with enough information to recreate the issue described in the report. In 7 cases, the app interacted with a remote server outside of my control, and the communication protocol between app and server had changed, preventing the app from running. In 3 cases, the issue depended on a specific hardware/software configuration that was not available to us. In the last case, part of the issue description in the report was written in a language other than English, preventing us from fully understanding the issue.

6.3.2 Results

RQ1 To answer RQ1, I applied the technique to the set of 62 reproducible issues discussed above. Overall, YAKUSU was able to successfully generate a test case for 37 of these 62 issues. (a success rate of 59.7%). In order to consider a generated test successful, I manually checked whether (1) the actions in the generated test matched the steps that a human would perform and (2) the stack trace generated by the test case corresponded to the one in the bug report (if one was present). Tables 6.1 and 6.2 report details and results for 25 of the cases, as 12 of the 37 issues simply required to launch the app and were trivial to reproduce. It is worth mentioning that certain apps required some setup (*e.g.,* user authentication) before they could be tested. As it is typical in these cases, and during testing in general, in the evaluation I provided this once-per-app setup information to YAKUSU.

Table 6.1 reports the identifier of the issue (*ID*), the name of the app (*Name*), the GitHub issue number (*Issue*), the lines of code in the app (*LOC (K)*), and the number of stars for the app on GitHub (*Stars*), which is a measure of popularity. Issues are ordered by their identifiers, whose values correspond to the order in which I (randomly) selected them.

On Table 6.2, The columns labeled *Actions* show the number of UI actions required to manually reproduce the issue described in the bug report, which consist of the sum of the number of actions that are explicitly (A_e) and implicitly (A_i) documented in the bug

Table 6.1: Benchmarks for which YAKUSU translated the bug report into a test case. For each benchmark considered: *ID* = identifier; *Name* = name; *Issue* = identifier of the GitHub issue considered; *LOC(K)* = # lines of code (thousands); *Stars* = # stars on GitHub;

ID	*Name*	*Issue*	*LOC (K)*	*Stars*
01	TACHIYOMI	880	38	1603
03	TWIDERE	738	141	1672
05	SIGNAL	6660	125	9803
08	REDREADER	516	42	830
14	SILENCE	557	109	871
23	K-9 MAIL	1910	136	3868
25	NEXTCLOUD	883	78	768
27	BUTTERKNIFE	46	5	165
35	ODK COLLECT	360	49	292
39	PIX-ART MESSENGER	127	58	30
40	YALP STORE	204	17	960
50	OCREADER	48	13	49
51	WORDPRESS	5497	180	1758
52	SIGNAL	6924	126	9803
57	OPEN EVENT	1402	18	344
68	TAGMO	12	32	663
69	ANKIDROID	4586	101	1231
73	K-9 MAIL	2612	137	3868
74	CLUTTR	2	13	9
78	NEXTCLOUD	850	74	768
84	K-9 MAIL	2019	136	3868
95	MIFOSX	734	65	85
96	SCREENRECORDER	25	7	62
97	NEXTCLOUD	1061	81	768
99	FLASHCARDS	13	5	8

report. These latter are actions that were not specified in the bug report but that need to be performed to reproduce the issue. As the table shows, for 9 of the 25 issues (rows with A_i¿0), at least one implicit action was involved in the reproduction task. In particular, for MIFOSX, the number of implicit actions was higher than number of explicit actions. (The implicit actions, in this case are "opening a sliding menu", "clicking on any element of a list", and "clicking on the element of the UI displaying an icon".) It is important to stress that the presence of implicit actions is far from rare, and generating tests for bug reports that involve such actions is particularly challenging. Therefore, the fact that YAKUSU was able to suitably handle these cases indicates its effectiveness and potential usefulness.

The columns labeled *Steps* describe the number of abstract steps generated by the technique. Specifically, they show the number of abstract steps (AS) and the number of generic abstract steps (AS_g) generated by YAKUSU for each issue. The sum of these two numbers, AS and AS_g, corresponds to the number of abstract steps provided as input to the search

Table 6.2: Details on the process of running YAKUSU. For each benchmark considered: *ID* = identifier; A_e = # explicit actions in the issue; A_i = # implicit actions in the issue; *AS* = # abstract steps generated; AS_g = # generic abstract steps generated; S_g = # states generated; S_p = # states processed; *H* = # heuristics used in the successful state; *R* = # random actions generated in the successful state; TC_s = number of statements in the test; TC_a = number of additional statements in the test; T_{oe} = time taken by the ontology extraction phase; T_{bra} = time taken by the bug report analysis phase; T_{uias} = time taken by the UI actions search phase.

ID	*Actions*		*Steps*		*Search*				*Tests*		*Cost*		
	A_e	A_i	*AS*	AS_g	S_g	S_p	*H*	*R*	TC_s	TC_a	T_{oe}	T_{bra}	T_{uias}
01	4	1	-	4	10	1	1	-	5	-	54s	18s	1m12s
03	2	-	2	-	12	1	-	-	2	-	2m34s	15s	1m07s
05	1	-	1	-	2	1	-	-	1	-	1m23s	17s	6m57s
08	2	-	2	2	5	1	-	-	4	2	25s	19s	1m25s
14	1	-	1	-	2	1	-	-	1	-	43s	15s	5m08s
23	3	-	3	-	8	1	-	-	3	-	32s	16s	1m00s
25	1	2	1	-	1	1	1	1	3	-	34s	16s	2m08s
27	1	-	1	-	3	1	-	-	1	-	13s	16s	09s
35	2	-	1	4	11	3	-	-	2	-	30s	18s	3m51s
39	3	-	3	-	2	1	-	-	3	-	43s	15s	31s
40	5	-	4	1	11	1	-	-	5	-	16s	17s	2m25s
50	3	-	3	-	8	2	-	-	3	-	22s	17s	1m33s
51	3	-	2	1	2	1	-	-	3	-	1m35s	16s	3m18s
52	2	1	2	-	3	1	-	4	6	3	1m16s	17s	18m10s
57	1	1	1	1	3	2	1	-	2	-	36s	20s	2m48s
68	1	1	1	1	10	1	-	3	4	2	17s	16s	31s
69	5	-	5	-	8	1	-	-	5	-	29s	17s	28m55s
73	2	-	2	-	6	3	-	-	2	-	35s	18s	3m22s
74	2	-	2	-	6	1	-	-	2	-	26s	15s	24s
78	2	1	2	-	1	1	1	-	3	-	32s	15s	2m20s
84	1	-	-	1	2	1	-	-	1	-	32s	17s	45s
95	2	3	2	-	3	1	1	2	5	-	39s	18s	1m25s
96	3	2	3	-	1	1	2	-	5	-	15s	17s	54s
97	1	1	-	2	4	2	1	-	2	-	37s	23s	9m28s
99	3	-	2	1	6	2	-	3	4	1	15s	18s	58s

phase of the technique, which is responsible for generating an actual test case. This sum can be different from the sum of actions for two reasons. First, implicit actions are not translated into abstract steps, as they are not present in the bug report. Second, there could be generic abstract steps that might not actually describe a UI action. As an example, consider the bug report associated with the OPEN EVENT issue, which includes the following sentence: "Take the pull of the latest code". YAKUSU translates this sentence into a generic abstract step that does not describe a specific action on the UI. As explained in Section 6.2, the technique handles this situation by forking an execution that discards this step. Conversely, for seven other issues (*i.e.,* issues 01, 35, 40, 51, 84, 97, and 99), generic abstract

steps are essential to reproduce the issue. This number suggests that YAKUSU is able to handle actions in bug reports that are expressed at different levels of abstraction. It is also worth noting that seven of the bug reports reproduced by YAKUSU did not contain header "reproduce", which provides initial evidence that YAKUSU is able to handle less structured bug reports.

The columns labeled *Search* provide information on the outcome of the UI-actions-search phase of the technique. Specifically, they show statistics of the search that led to the generation of the test case corresponding to the input abstract steps: number of states generated (S_g), number of states processed (S_p), number of heuristics used (H), and number of random UI actions in the generated test case (R). In 16 cases, the search never had to select a different state to explore other than the current state (entries with S_g¿1 and S_p=1). These are cases of quick successful runs, in which the technique did not have to start the app again and restore its state (lines 6-7 from Algorithm 5). In another 6 cases, conversely, the technique had to explore other states to find the list of UI actions that satisfied all the input abstract steps (entries with S_g¿1 and S_p¿1); these cases highlight the importance of tracking and exploring multiple states during the search. As columns H and R show, in creating test cases, YAKUSU used at least one of its heuristics and generated at least one random action in seven and six cases, respectively.

Finally, the columns labeled *Tests* show the number of required (TC_s) and non-required (TC_a) statements in the generated test cases. As the table shows, YAKUSU generated non-required statements in only four cases.

To have an understanding of the causes behind false negatives, I investigated the 25 (*i.e.,* 62-37) cases in which YAKUSU was unable to successfully generate a test case and grouped them into four categories. Category "Cat 1" (7 cases) contains reports with actions that need to be performed outside of the app (*i.e.,* actions on the UI of the operating system). To address these cases, I could extend the ontology generated by YAKUSU with system actions. Category "Cat 2" (3 cases) includes reports with actions on UI elements whose properties

cannot be introspected at runtime (*e.g.,* custom views [58]). I plan to explore how to make such properties available at runtime through app instrumentation. Category "Cat 3" (11 cases) contains reports in which a single step corresponds to multiple implicit UI actions. The sentence "After a while browsing through folders app crash [sic] immediately" [23], for instance, should be translated to a sequence of clicks on various folders, but YAKUSU fails to interpret the sentence correctly. I plan to investigate ways to handle these cases by leveraging related work (*e.g.,* [125, 126, 29, 152]) and incorporating additional domain knowledge, possibly on demand. Category "Cat 4" (4 cases) includes reports with actions that involve multi-touch gestures (*e.g.,* pinch zoom). YAKUSU could handle these actions through suitable, albeit extensive, engineering. I did not observe false positives, that is, successfully generated test cases that did not lead to the failure described in the bug report. If these cases were to occur, in the context of bug reports describing crashes, YAKUSU could address this issue by filtering out non-crashing tests.

In summary, I consider the results for RQ1 encouraging. Despite the limitations presented above, which can be addressed as discussed, YAKUSU was already able to generate test cases for a majority of bug reports while generating no spurious tests.

RQ2 To answer RQ2, I measured the time taken to run each phase of the technique on a MacBook Pro with 2.8 GHz i7 processor and 16GB of RAM. Table 6.2 shows, for each issue considered, the time required to extract the ontology (T_{oe}), perform bug report analysis (T_{bra}), and explore the app to generate a test case (T_{uias}). The times in the table are expressed in minutes (m) and seconds (s).

As the table shows, the UI-actions-search phase is where YAKUSU spent most of its time, followed by the ontology-extraction phase, and then the bug-report-analysis phase. The technique generates a test case in less than five minutes overall in most cases, with the average and median times being 5m00s and 2m58s, respectively. In only two cases, the execution time was above ten minutes. In particular, ANKIDROID (issue 69) is the case

in which YAKUSU takes the longest time to generate a test case: 29m41s. The execution time for this benchmark is dominated by the UI actions search phase of the technique, which is 28m55s. This value is higher than the one associated with other benchmarks due to a higher number of computations of the cosine similarity value. (This operation is relatively expensive compared to other operations because it requires to issue a network request from the test device to get the similarity value.) Considering that the execution time is fairly low even in the worst case, I did not further investigate this issue, nor tried hard to optimize YAKUSU. In fact, these execution times suggest that YAKUSU could be used to monitor bug reports and generate test cases throughout the day, as opposed to overnight only. Moreover, for issues that cannot be translated successfully and may result in longer running explorations of the app states, developers could provide a suitable timeout.

6.3.3 Threats To Validity

As it is the case for most empirical evaluations, there are both external and construct threats to validity associated to the results I presented. In terms of external validity, the results might not generalize to other bug reports or apps. In particular, I only considered 100 bug reports. This limitation is an artifact of the complexity involved in manually building and setting up the infrastructure to run the app associated with a bug report. To mitigate this threat, I used randomly selected real-world bug reports from different apps. An additional threat could be posed by the fact that I used only open source apps in the evaluation. However, the evaluation includes apps such as K-9 MAIL, SIGNAL, and WORDPRESS, which have complex functionality, hundreds of widgets, and millions of users. I believe that, given the complexity of the apps I analyzed, YAKUSU should also be applicable to other types of apps. In terms of construct validity, there might be errors in the implementation of the technique. To mitigate this threat, I extensively inspected the results of the evaluation manually.

CHAPTER 7

PERFORMING API-USAGE UPDATES

This chapter discusses my technique that automatically updates an app's code to account for changes to the API of the underlying operating system for improving software quality after release. The technique, named APPEVOLVE, automatically updates API usages (i.e., any call of one or more methods to the API) by analyzing how developers of other apps performed corresponding changes. Automating this task is particularly challenging as updates from different developers do not always share the same set of operations. The rest of this chapter is structured as follows. Section 7.1 provides the problem statement and terminology used in the chapter, Section 7.2 illustrates a motivating example, Section 7.3 presents the technique, and Section 7.4 discusses the evaluation of APPEVOLVE.

7.1 Problem Statement and Terminology

Consider two versions of the API: *old API version* = $[m_1, ..., m_k]$ and *new API version* = $[m'_1, ..., m'_l]$. I define an *API usage* as any call of one or more methods in either the old or the new API versions. An API-usage change between old version and new version of the API can typically be described in terms of a mapping between one or more methods in the old version and one or more methods in the new version: $AU\text{-}C = [m_1, ..., m_p] \rightarrow [m'_1, ..., m'_q]$. These API-usage changes are the ones targeted by APPEVOLVE. To illustrate, consider the change for the Android API version 23 described at [57]. In this case, the API-usage change would be the mapping from method `NetworkInfo[] getAllNetworkInfo()` to methods `Network[] getAllNetworks()` and `NetworkInfo getNetworkInfo (Network network)`. Given an API-usage change $AU\text{-}C$, I define an *old API usage* (resp., *new API usage*) for $AU\text{-}C$ as a sequence of one or more method invocations that matches the left-hand side (resp., right hand side) of the $AU\text{-}C$ mapping. For the example I just provided,

an old API usage would be an invocation of `NetworkInfo[] getAllNetworkInfo()`, whereas a new API usage would be an invocation of `Network[] getAllNetworks()` followed by an invocation of `NetworkInfo getNetworkInfo (Network network)`. Considering an app A and an API-usage change AU-C, I use the term *update* to indicate the operation of updating an old API usage (AU) from AU-C in A to a new API usage (AU'), and call the updated app A'. I call each existing update an *update example*.

7.2 Motivating Example

To motivate my work, I use two examples (A_1 and A_2) derived from actual updates of real-world apps. The two examples perform the update of the old API usage $AU = [\texttt{getAllNetworkInfo}]$ (here and in the remainder of the paper I use the method name instead of its signature to represent an API usage for brevity). Method `getAllNetworkInfo` returns connection information of all network types supported by a device. The API usage was deprecated in API version 23 of the Android platform in favor of the new API usage $AU' = [\texttt{getAllNetworks}, \texttt{getNetworkInfo}]$ because `getAllNetworkInfo` does not support multiple connections of the same type. Figures 7.1a and 7.1b illustrate the update example from A_1 while Figures 7.2a and 7.2b from A_2. In all figures, lines starting with – indicate code removed by the update while + indicate added code.

Figure 7.1a shows the code in the old version of A_1 using AU. The code invokes `getAllNetworkInfo` at line 4 and iterates over the returned array of `NetworkInfo` objects at lines 6-9 to check whether the device is connected to a network. The new version of the code is in Figure 7.1b. In this version, the developer introduced a check at line 4 to determine the version of the platform on which the app is running. In case the app is running on the old version of the platform (lines 13-18), the developer checks whether the device is connected to a network similarly to how it was performed in the old version of the code. When the app is running on the new version of the platform (lines 5-11), the developer invokes `getAllNetworks` (from the new API) at line 5 and iterates over the array of

```
1  public boolean isNetworkAvailable(Context ctx) {
2    ConnectivityManager ctv = (ConnectivityManager)
3      ctx.getSystemService(Context.CONNECTIVITY_SERVICE);
4 -  NetworkInfo[] info = ctv.getAllNetworkInfo();
5 -  if (info != null) {
6 -    for (int i = 0; i < info.length; i++) {
7 -      if (info[i].getState() == NetworkInfo.State.CONNECTED) {
8          return true;
9        } }
10   }
11   return false;
12 }
```

(a) API-usage example before update in A_1.

```
1  public boolean isNetworkAvailable(Context ctx) {
2    ConnectivityManager ctv = (ConnectivityManager)
3      ctx.getSystemService(Context.CONNECTIVITY_SERVICE);
4 +  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
5 +    Network[] networks = ctv.getAllNetworks();
6 +    NetworkInfo networkInfo;
7 +    for (Network mNetwork : networks) {
8 +      networkInfo = ctv.getNetworkInfo(mNetwork);
9 +      if (networkInfo.getState().equals(NetworkInfo.State.CONNECTED)) {
10         return true;
11       }  }
12 +  } else {
13 +    NetworkInfo[] info = ctv.getAllNetworkInfo();
14 +    if (info != null) {
15 +      for (NetworkInfo anInfo : info) {
16 +        if (anInfo.getState() == NetworkInfo.State.CONNECTED) {
17 +          return true;
18 +        }  }  }
19   }
20   return false;
21 }
```

(b) API-usage example after update in A_1'.

Figure 7.1: API-usage example for A_1 and A_1'.

`Network` objects to determine whether the device is connected to a network (lines 7-11). To determine the connection status, the developer retrieves network information from a `Network` object invoking `getNetworkInfo`. The update example is characterized by two pieces of code (lines 13-18 and lines 5-11), each executing on different versions of the platform (and having access to different APIs). As pointed out by related work [94, 74], this coding practice is frequent in Android apps to account for issues generated by the fragmentation of the ecosystem [94, 74, 132]. This example also shows that updating API usages might require to use new methods, handle new parameters, and manipulate new return values.

```
1  public boolean isConnected(Context cont) {
2    ConnectivityManager conn = (ConnectivityManager)
3      cont.getSystemService(Context.CONNECTIVITY_SERVICE);
4 -  NetworkInfo[] info = conn.getAllNetworkInfo();
5 -  if (info != null) {
6 -    for (int i = 0; i < info.length; i++) {
7 -        if(info[i].isConnected()) {
8            return true;
9          }  }
10   }
11 -  Toast.makeText(cont, R.s.noNet, Toast.L_S).show();
12   return false;
13 }
```

(a) API-usage example before update in A_2.

```
1  public boolean isConnected(Context cont) {
2    ConnectivityManager conn = (ConnectivityManager)
3      cont.getSystemService(Context.CONNECTIVITY_SERVICE);
4 +  if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
5 +    Network[] networks = conn.getAllNetworks();
6 +    NetworkInfo networkInfo;
7 +    for (Network mNetwork : networks) {
8 +      networkInfo = conn.getNetworkInfo(mNetwork);
9 +      if(networkInfo.isConnected())) {
10 +       Log.d("Net","NAME:"+networkInfo.getTypeName());
11         return true;
12       }  }
13 +  } else {
14 +    NetworkInfo[] info = conn.getAllNetworkInfo();
15 +    if (info != null) {
16 +      for (NetworkInfo anInfo : info) {
17 +        if(anInfo.isConnected()) {
18 +          Log.d("Net","NAME:"+anInfo.getTypeName());
19 +          return true;
20 +        }  }  }
21   }
22 +  Toast.makeText(cont, cont.getString(R.s.noNet), Toast.L_S).show();
23   return false;
24 }
```

(b) API-usage example after update in $A_2{}'$.

Figure 7.2: API-usage example for A_2 and $A_2{}'$.

As it is possible to observe from Figures 7.2a and 7.2b, the update of AU in A_2 is similar to the one performed in A_1 as both updates introduce a condition check to determine the version of the platform on which the app is running and both invoke `getAllNetworks` and `getNetworkInfo` from the new API in one branch of the condition and `getAllNetwork-Info` in the other. At the same time, the two updates also have some differences. In fact, the code in A_1 checks the connection status of a `NetworkInfo` object by comparing the result obtained by invoking `getState` against `CONNECTED` (line 7 in Figure 7.1a and lines 9 and 16 in Figure 7.1b) while A_2 uses `isConnected` to perform the same task. The update

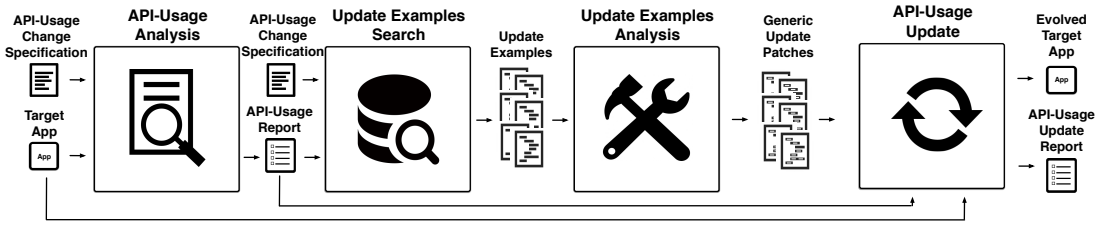


Figure 7.3: High-level overview of APPEVOLVE.

in A_2 also logs information about the type of network being connected (lines 10 and 18 in Figure 7.2b) and makes changes that are not related to the API usage (line 11 in Figure 7.2a and line 22 in Figure 7.2b).

The motivating example highlights the potential of using update examples to updated API usages but also presents some of the challenges in automatically doing so, as the updates share some commonalities but also have differences. An example of commonality is the structure based on the condition check on the version of the API, while examples of differences are: use of `getState` and `CONNECTED` vs. use of `isConnected` and also inclusion of additional statements such as `Log.d` that are actually unrelated to the update.

7.3 Technique

In this section, I present APPEVOLVE, a technique for automatically performing API-usage updates based on update examples. The technique targets Android apps and updates related to changes in the API of the Android platform. The basic idea behind APPEVOLVE is to update an API usage in a target app by leveraging how developers of other apps updated the given usage in their apps.

Figure 7.3 provides an overview of APPEVOLVE's workflow and show its four main phases. Given a target app and a specification of the API-usage changes as inputs, in its *API-usage analysis* phase, the technique analyzes the source code of the app to identify API usages that should be changed and store this information in the *API-usage report*. The *update examples search* phase uses this information together with the specification to look for API-usage updates in existing code bases. The *update examples analysis* phase pro-

cesses the examples identified to generalize them, identify their commonalities, and rank them based on the proximity to their shared commonalities. In this process, the examples are transformed into *generic update patches*. Finally, the ***API-usage update*** phase leverages the generic update patches to change the API-usage locations reported by APPEVOLVE's first phase and validates them using differential testing. The final outputs of the technique are an evolved app and an API-usage update report documenting the changes in the app.

7.3.1 API-Usage Analysis

This phase takes as inputs the source code of a target app (T) and a specification of the API-usage changes ($AU\text{-}CS$), and creates an API usage report containing the location in T of old API usages that should be updated. The content of $AU\text{-}CS$ is a set of API-usage changes ($AU\text{-}C_1$, ..., $AU\text{-}C_n$). An API-usage change follows the specification of Section 7.1. In the case of the motivating example, $AU\text{-}CS$ = ($AU \rightarrow AU'$), where the old API usage (AU) is [`getAllNetworkInfo`] and the new API usage (AU') is [`getAllNetworks`, `getNetworkInfo`].

At a high level, APPEVOLVE identifies old API usages that should be updated by checking if they can execute while T is running on the new version of the OS (and, therefore, its new API). (Apps can run on multiple versions of the API but not necessarily all of their code can run on the new version of the API [94, 74].) The technique computes this information by statically analyzing the source code of T. First, APPEVOLVE computes the set of API versions (S_{API}) on which statements in T can execute. The technique does so by leveraging an inter-procedural dataflow analysis defined in related work [74]. Second, the technique performs an intra-procedural analysis of methods declared in T to identify old API usages and checks if calls to methods in them can execute on the new version of the API (based on S_{API}). If this is the case, such API usages require update and APPEVOLVE stores them in the API-usage report together with the location in T of the calls of their composing methods.

Algorithm 6: Search for update examples in APPEVOLVE.

```
   Input : AU: API usage in old version of API
           AU': API usage in new version of API
           chi: code hosting infrastructure based on version-control system
   Output: ues_{AU-C}: update examples for AU-C = AU → AU'
1  begin
2    ues_{AU-C} = ∅
3    kws = COMPUTE-KEYWORDS(AU')
4    files = FIND-FILES(kws, chi.GET-INDEX())
5    foreach f ∈ files do
6      cb = f.GET-CODE-BASE()
7      versions_f = cb.GET-VERSIONS(f)
8      foreach v_f ∈ versions_f do
9        v_n = v_f
10       v_o = FIND-PREVIOUS-VERSION(versions_f, v_n)
11       if v_o == null then
12         continue
13       diff = COMPUTE-DIFFERENCES(v_o, v_n)
14       lines_r = diff.GET-REMOVED()
15       lines_a = diff.GET-ADDED()
16       if ¬(v_o.lines.USES(AU')) ∧ lines_r.USES(AU))
17        ∧ (lines_a.USES-WITH-CHECK(AU, AU'))
18        ∧ SAME-CONTAINING-METHOD(v_o, AU, v_n, AU') then
19         sig_o = v_o.GET-CONTAINING-METHOD-SIGNATURE(AU)
20         sig_n = v_n.GET-CONTAINING-METHOD-SIGNATURE(AU')
21         ue_{AU-C} = UP-EX(cb, v_o, sig_o, v_n, sig_n, AU, AU')
22         ues_{AU-C}.ADD(ue_{AU-C})
23   return ues_{AU-C}
```

7.3.2 Update Examples Search

This phase identifies a set of update examples for each old API usage reported in APPEVOLVE's first phase by looking at how other developers updated the corresponding API usage in their apps. The technique does so by analyzing the version control history and the source code of other apps' code bases. The final output of this phase is a set of update examples for each old API usage (AU) that requires update in T. I detail how APPEVOLVE automatically identifies update examples with the help of Algorithm 6.

The algorithm takes as inputs an old API usage (AU), the corresponding new API usage (AU'), and the location of a code hosting infrastructure (chi) where code bases for other apps are publicly accessible. The output of the algorithm is a set of update examples (ues_{AU}) for AU. At a high level, the algorithm has the goal to identify one or methods that updated AU to AU'.

The algorithm starts with an empty set of examples (line 2) and identifies code bases that could contain examples by searching for files in chi that use AU'. The algorithm accomplishes this task at lines 3-4 where it performs a textual search on the files. The search (FIND-FILES) is based on a set of keywords extracted from AU' and an index built on the

content of the files in chi. This step enables APPEVOLVE to efficiently consider the content of a large number of code bases and quickly discard irrelevant ones (using the intuition that if a developer performed an update, the update should be present in the latest version of a file). The set of keywords (kws) contains: (1) the name, (2) parameter types, and (3) the declaring class for each method in AU'. These terms are likely to appear in files using AU'. For the motivating example, $kws = \{$`getAllNetworks`, `ConnectivityManager`, `getNetworkInfo`, `Network` $\}$.

At this point, the algorithm processes each file (f) resulting from the search to identify the ones whose history contains an update of AU to AU' (lines 5-22). To do so, the algorithm processes each version (v_n) of f with its preceding version (v_o) starting from the most recent version for f. The algorithm assumes that updates occur between two contiguous versions. Given these two versions, the algorithm computes their differences [121] and extracts removed lines ($lines_r$) from v_o and added lines ($lines_a$) in v_n.

The core part of the algorithm is described at lines 16-22 where it checks if an update of AU to AU' between v_o and v_n is present. The algorithm performs this task by: (1) checking that in v_o there is no use of AU', (2) looking for a method in v_o whose removed lines are using AU, and (3) identifying the corresponding method in v_n that added new lines using AU and AU' (in different branches of a condition checking the version of the API). The algorithm searches for methods in v_n also using AU as it aims to find examples that perform backward compatible updates. This is common practice when updating Android apps as reported by related work [74]. If the conditions of the analysis (lines 16-18) are met, the algorithm considers the update between the method in v_o (sig_o) and v_n (sig_n) to represent an update example for AU and adds this example to the list of examples for the API usage (line 21). In the motivating example, both updates would be considered as examples as they do not use $AU' =$ [`getAllNetworks`, `getNetworkInfo`] in their old version, removed lines in their old version contain $AU =$ [`getAllNetworkInfo`], and added lines in their new version contain AU and AU' in opposite branches of `SDK_INT` check.

Algorithm 7: Analysis of update examples in APPEVOLVE.

```
   Input : ues_{AU-C}: update examples for AU-C = AU → AU'
   Output: gups_{AU-C}:: ordered list of generic update patches
1  begin
2    //create generic update patches
3    gups_{AU-C} = []
4    foreach ue_{AU-C} ∈ ues_{AU-C} do
5      ast_o = BUILD-AST(ue_{AU-C}.sig_o)
6      ast_n = BUILD-AST(ue_{AU-C}.sig_n)
7      edits = COMPUTE-EDITS(ast_o, ast_n)
8      redits = FIND-RELATED-EDITS(edits, ast_o, ue_{AU-C}.AU, ast_n, ue_{AU-C}.AU')
9      gedits = []
10     foreach redit ∈ redits do
11       plcmnt, plcmnt_2 = COMPUTE-PLACEMENT(redit, redits)
12       abstr = COMPUTE-ABSTRACTION(redit)
13       gedit = GENERIC-EDIT(redit, plcmnt, plcmnt_2, abstr)
14       gedits.ADD(gedit)
15     cvars = FIND-CONTEXT-VARIABLES(ast_n, gedits)
16     ANNOTATE-CONTEXT-VARIABLES(cvars, ast_o, AU)
17     gup_{AU-C} = GENERIC-UPDATE-PATCH(gedits, cvars)
18     gups_{AU-C}.ADD(gup_{AU-C})
19   //sort generic update patches based on proximity to core of update
20   caseq = COMPUTE-CORE(gups_{AU-C})
21   foreach gup_{AU-C} ∈ gups_{AU-C} do
22     cprox = COMPUTE-CORE-PROXIMITY(gup_{AU-C}, caseq)
23     gup_{AU-C}.SET-CORE-PROXIMITY(cprox)
24   gups_{AU-C} = ORDER-BY-CORE-PROXIMITY(gups_{AU-C})
25   return gups_{AU-C}
```

The algorithm stores the following information about the example: (1) the code base (cb) and its version history information, (2) the version of f before the update (v_o), (3) the method signature in v_o containing AU (sig_o), (4) the version of f after the update (v_n), (5) the method signature in v_n containing AU and AU' (sig_n), and (6) the API usages associated with this update (AU and AU'). The algorithm terminates by returning the list of update examples ($ues_{AU\text{-}C}$) for the API usage under analysis.

7.3.3 Update Examples Analysis

This phase takes as inputs the update examples identified in the previous phase and aims to translate them into *generic update patches* so that they can be applied to the target app. In this phase, APPEVOLVE processes update examples associated with a specific API-usage change together and performs two tasks: (1) it generalizes the examples into generic update patches and (2) it orders the patches based on how closely related they are to the commonalities of the update shared across examples. The technique performs the second task to prioritize patches of examples that best capture the essence of the update. I use Algorithm 7 to describe the two tasks.

```
I if Build.VERSION.SDK_INT>=Build.VERSION_CODES.M
I Network[] networks=ctv.getAllNetworks()
I NetworkInfo networkInfo
I for Network mNetwork:networks
M NetworkInfo[] info=ctv.getAllNetworkInfo()
M if info!=null
I networkInfo=ctv.getNetworkInfo(mNetwork)
U if info[i].getState()==CONNECTED
  if networkInfo.getState().equals(CONNECTED)
M if networkInfo.getState().equals(CONNECTED)
I for NetworkInfo anInfo:info
I if anInfo.getState()==CONNECTED
I return true
D for int i=0 i<info.length i++
```

(a) Edits for update example provided by A_1.

```
I if Build.VERSION.SDK_INT>=Build.VERSION_CODES.M
U Toast.makeText(cont,noNet,L_S).show()
  Toast.makeText(cont,conn.getString(noNet),L_S).show()
I Network[] networks=conn.getAllNetworks()
I NetworkInfo networkInfo
I for Network mNetwork:networks
M NetworkInfo[] info=conn.getAllNetworkInfo()
M if info!=null
I networkInfo=conn.getNetworkInfo(mNetwork)
U if info[i].isConnected()
  if networkInfo.isConnected()
M if networkInfo.isConnected()
I Log.d("Net","NAME:"+networkInfo.getTypeName())
I for NetworkInfo anInfo:info
I if anInfo.isConnected()
I Log.d("Net","NAME:"+anInfo.getTypeName())
I return true
D for int i=0 i<info.length i++
```

(b) Edits for update example provided by A_2.

Figure 7.4: Edits for update examples provided by A_1 and A_2.

Generic Update Patch Generation The algorithm generalizes the input update examples ($ues_{AU\text{-}C}$) related to an API-usage change at lines 3-18. The algorithm begins by initializing the list of generic update patches ($gups_{AU\text{-}C}$) to be empty and then processes each update example ($ue_{AU\text{-}C}$). APPEVOLVE starts the task of generalizing $ue_{AU\text{-}C}$ by identifying a list of edit operations that transforms the method (represented by sig_o) from the old version of the code in $ue_{AU\text{-}C}$ (containing AU) to the method (represented by sig_n) from the new version of the code (containing AU and AU'). The algorithm encodes these edits ($edits$) in terms of tree operations performed on the abstract syntax trees built from the methods (ast_o and ast_n). Edits are computed based on a technique defined in related work [42, 114]. This technique identifies an ordered list of tree operations to transform one abstract syntax tree (AST) into the other and operates on the nodes representing statements in ASTs. There are four types of edit operations: (1) INSERT(sn_1, sn_2, i) that adds statement node sn_1 as i^{th} child of statement node sn_2, (2) MOVE(sn_1, sn_2, i) that moves statement node sn_1 from its current position and adds it as i^{th} child of statement node sn_2, (3) UPDATE(sn_1, sn_2) that updates statement node sn_1 to be statement node sn_2, and (4) DELETE(sn_1) that deletes statement node sn_1 from the tree. For each type of edit, I refer to sn_1 as the statement *affected* by the edit and denote it with as. Figures 7.4a and 7.4b provide the edits for the update examples from A_1 and A_2, respectively (I provide a summarized version).

At this point, the algorithm identifies edits that are related to AU and AU' using an intra-procedural dependency analysis (FIND-RELATED-EDITS at line 8). The algorithm selects the edits that are influencing or are influenced by the values involved in method calls from AU and AU' using backward and forward dependency analysis. The analysis uses both ast_o and ast_n to compute dependencies as certain edits might operate only on statements in one of the two trees (*e.g.,* `D for int i=0 i<info.length i++` from Figures 7.4a and 7.4b). This operation identifies the set of related edits ($redits$ at line 8) for the update. The technique performs this operation to disregard edits that are not related to the API-usage update. In the motivating example, APPEVOLVE determines that all edits in Figure 7.4a are related to the update while all edits but `U Toast.makeText...` in Figure 7.4b are considered as such.

The edits obtained so far are dependent on the specific example from which they were computed. At this point, the algorithm generalizes them so that they can be used to update AU in the methods of the target app. To do so, the algorithm iterates over $redits$ (lines 10-14), and translates each edit into a generic edit ($gedit$). A generic edit is composed by the original edit ($redit$), the placement ($plcmnt$) of the statement as affected by $redit$, and an abstraction ($abstr$) of as. At the end of this loop, the algorithm obtains an ordered list of generic edits ($gedits$). Placement $plcmnt$ is expressed in terms of an ancestor statement $ancs$ and a predecessor statement $preds$. (Not every as has an $ancs$ or a $preds$.)

$ancs$ and $preds$ are computed by analyzing the statements affected by edits in $redits$ that occur before the edit under analysis, I denote such edits as $predits$. $ancs$ is computed by analyzing parent relations in the AST and corresponds to the first statement in the relation chain starting at as that is affected by an edit in $predits$. $pred$ is the latest statement appearing before as in a postorder traversal of the AST that: (1) is not present in the subtree rooted at as and (2) it is affected by and edit in $predits$. In the case of MOVE edits, the technique also computes $plcmnt_2$ containing $ancs$ and $preds$ for the new position of as. The computed information about $ancs$ and $preds$ is necessary as the original

position of a statement affected by an edit is based on the AST on which it was computed and will not be applicable in different ASTs. In Figure 7.4a, edit `I NetworkInfo networkInfo` has `I if Build.VERSION.SDK_INT>= Build.VERSION_CODES.M` as *ancs* and `I Network[] networks = ctv.getAllNetworks()` as *pred*. Abstraction *abstr* is a representation of a statement in which its variables are replaced by their types. APPEVOLVE uses this abstraction in a later stage of the algorithm to compute commonalities of the update across examples. As an example, the abstraction for statement `if info!=null` is `if NetworkInfo!=null`.

At this point (line 15), the algorithm extracted an ordered list of generic edits from an update example and proceeds forward by identifying the variables (and their types) that are used by statements affected by the edits but are not defined in the statements (the algorithm does so only for statements from ast_n). I refer to such variables as *context variables* ($cvars$). In the case of the motivating example, the edits associated with the example from A_1 have `ctv` of type `ConnectivityManager` as the only context variable. Context variables are further processed (ANNOTATE-CONTEXT-VARIABLES) to identify the ones that are directly used by method invocations in AU. These variables are annotated and this information will be used by APPEVOLVE when applying a patch to the target app. For example, variable `ctv` from A_1 will be annotated because it is used by method `getAllNetworkInfo`. Context variables and generic edits define the content of a generic update patch ($gup_{AU\text{-}C}$). Context variables together with the characteristics of the edits define the *applicability* of a patch. I detail the concept of applicability in Section 7.3.4 when I describe the process of applying generic update patches to the target app. Finally, the patch generated from the update example is added to the list of patches ($gup_{AU\text{-}C}$).

Generic Update Patch Prioritization Once all update examples are translated into generic update patches, the algorithm orders them (lines 20-24) based on how related they are to the commonalities of the update shared across all patches (and consequently the examples).

```
I if SDK_INT>=M
I Network[] $T=$T.getAllNetworks()
I NetworkInfo $T
I for Network $T:$T
M NetworkInfo[] $T=$T.getAllNetworkInfo
M if $T!=null
I $T=$T.getNetworkInfo($T)
U if $T[$T].getState()==CONNECTED
M if $T.getState().equals(CONNECTED)
I for NetworkInfo $T:$T
I if $T.getState()==CONNECTED
I return true
D for int $T=0 $T<$T $T++
```

(a) Edit abstractions for update A_1.

```
I if SDK_INT>=M
I Network[] $T=$T.getAllNetworks()
I NetworkInfo $T
I for Network $T:$T
M NetworkInfo[] $T=$T.getAllNetworkInfo
M if $T!=null
I $T=$T.getNetworkInfo($T)
U if $T[$T].isConnected()
M if $T.isConnected()
I Log.d("Net","NAME:"+$T.getTypeName())
I for NetworkInfo $T:$T
I if $T.isConnected()
I Log.d("Net","NAME:"+$T.getTypeName())
I return true
D for int $T=0 $T<$T $T++
```

(b) Edit abstractions for update A_2.

```
I if SDK_INT>=M
I Network[] $T=$T.getAllNetworks()
I NetworkInfo $T
I for Network $T:$T
M NetworkInfo[] $T=$T.getAllNetworkInfo
M if $T!=null
I $T=$T.getNetworkInfo($T)
I for NetworkInfo $T:$T
I return true
D for int $T=0 $T<$T $T++
```

(c) Common edit abstraction subsequence between update examples A_1 and A_2.

Figure 7.5: Edit abstractions for the examples provided by A_1 and A_2.

I define commonality in terms of the longest subsequence of edits that is shared across all patches. I refer to this subsequence as the *core* of the update. I measure how closely related is a patch to the core by dividing the number of edits from the core by the number of edits in the patch. I refer to this value as *core proximity*. I translate the problem of finding the longest subsequence shared across all patches into the multiple longest common subsequence problem (MLCS) [69] and solve the problem to obtain the subsequence. Because patches from different examples might operate on variables with different names, I use an abstraction of each edit to encode patches into the MLCS problem. The abstraction for an edit is a string composed by the type of the edit followed by the abstraction $abstr$ of the statement affected by the edit. Figures 7.5a and 7.5b show the abstract edit sequences computed for the patches from A_1 and A_2, respectively (in the figures types are replaced by `$T`). The longest subsequence shared across the two patches is presented in Figure 7.5c.

After solving the MLCS problem (COMPUTE-CORE at line 20) the algorithm computes the core proximity value for each patch and orders patches (ORDER-BY-CORE-PROXIMITY) based on the value. In the case of the motivating example, the core proximity value associated with the patch computed from A_1 is 0.77 and the value associated with the patch computed from A_2 is 0.67. The two values show how the patch from A_1 is closer to the core as it does not contain additional logging statements (`Log.d`). At the same time the patch from A_1 does not have maximum value (1) as the two patches use different state-

Algorithm 8: API-usage update in APPEVOLVE.

```
   Input : T: target app
           AU: old API-usage that requires update
           loc_AU: location of the old API-usage in T
           gups_AU-C: generic update patches for AU-C = AU → AU'
   Output: T': target app with performed API-usage change AU-C
           ur_AU-C: API-usage update report for AU-C
 1 begin
 2   ur_AU-C = []
 3   foreach gup_AU-C ∈ gups_AU-C do
 4     sig_t = T.GET-CONTAINING-METHOD-SIGNATURE(loc_AU)
 5     ast_t = BUILD-AST(sig_t)
 6     foreach stmt ∈ ast_t.PREORDER() do
 7       svars = FIND-VARIABLES-IN-SCOPE(ast_t, stmt)
 8       mpngs = COMPUTE-MAPPINGS(svars, gup_AU-C.cvars, ast_t, AU)
 9       foreach mpng ∈ mpngs do
10         ast'_t = APPLY-UPDATE-PATCH(ast_t, stmt, mpng, gup_AU-C)
11         if ast'_t != null then
12           T' = T.REPLACE(CODE(ast_t), CODE(ast'_t))
13           if VALIDATE(T, T') then
14             ur_AU-C .ADD(VAL-UPDATE(CODE(ast_t), CODE(ast'_t)))
15             return T', ur_AU-C
16           else
17             ur_AU-C .ADD(APPL-UPDATE(CODE(ast_t), CODE(ast'_t)))
18   T' = T
19   return T', ur_AU-C
```

ments (`getState` and `CONNECTED` vs. `isConnected`). I decide to rank patches instead of merging them to avoid disregarding statements that are necessary to perform the update. The ordered list of patches $gups_{AU\text{-}C}$ is the final output of the algorithm.

7.3.4 API-Usage Update

The last phase of the technique applies generic update patches to old API usages in the target app and validate them using differential testing. The technique updates old API usages detailed in the API-usage report one at a time and does so to ensure that, during the validation process, updates for different API usages do not interfere with each other. I detail how this phase updates and validates an old API usage in the target app with the help of Algorithm 8.

The algorithm takes as inputs: (1) the target app (T), (2) the API usage that should be updated (AU), (3) the location of AU in T (loc_{AU}), and (4) the generic update patches ($gups_{AU\text{-}C}$) for the API-usage change. The algorithm produces as outputs: an updated target app T' and an API-usage update report ($ur_{AU\text{-}C}$) where changes (for the specific occurrence of AU) are documented to the developer of T. The algorithm begins by initializing $ur_{AU\text{-}C}$ to be empty and then processes patches starting from the patch that is closest

to the core of the update (positioned at the top of the list) at lines 3-17.

At a high level, the algorithm first determines whether a patch ($gup_{AU\text{-}C}$) is applicable to AU at loc_{AU} in T, and if this is the case, it validates the updated target app. Applicability of a patch is determined by two factors. First, context variables associated with the patch need to be in scope where the patch is applied. Second, generic edits should be successfully applied (in their entirety) to the method in T using AU. The algorithm validates an applicable patch using differential testing. If a test suite ts for T is available and ts exercises AU without failures, I use ts for the validation process. Otherwise, the algorithm builds ts by performing random input generation (but other input generation strategies are also possible) on T while executing on the old version of the API and recording generated inputs in ts. If input generation exposes a failure in T or does not exercise AU, the validation process terminates unsuccessfully as it is not possible to define a valid behavior of AU. To validate T', the algorithm executes ts while T' is running with the new version of the API and also while T' is running with the old API (to ensure backward compatibility). If no failures are reported by this process, the update performed in T' is considered as valid.

When determining applicability of a generic update patch $gup_{AU\text{-}C}$, the algorithm first builds the AST (ast_t) of the method containing AU based on loc_{AU} and then iterates (lines 6-17) over the statement nodes ($stmt$) in ast_t to find the point in which to apply $gup_{AU\text{-}C}$. The algorithm determines if $gup_{AU\text{-}C}$ can be applied at the point in ast_t represented by $stmt$ by computing the variables in scope (FIND-VARIABLES-IN-SCOPE) and identifying whether there is a mapping between context variables ($gup_{AU\text{-}C}.cvars$) and variables in scope ($svars$). The algorithm finds whether there is a mapping between the two sets by translating this task into an instance of the assignment problem [127] and considers as mappings all solutions of minimum cost. APPEVOLVE uses the algorithm proposed in related work [128] to solve the assignment problem. APPEVOLVE instantiates the algorithm by expressing the cost of mapping a context variable to a scope variable of the same type with the minimum cost of zero and maximum cost otherwise. All assign-

ment solutions of cost zero are considered as a plausible mappings between $gup_{AU\text{-}C}.cvars$ and $svars$. APPEVOLVE adds additional constraints on the assignment problem based on the variables in $gup_{AU\text{-}C}.cvars$ that are annotated as being used by method invocations from AU and corresponding variables in method invocations from AU in T that are part of $svars$. It does so by forcing the mapping between such variables. These additional constraints help APPEVOLVE by limiting the number of mappings when the number of compatible variables in scope is high. If no mapping is possible, the algorithm analyzes the variables in scope at the next location in the AST.

After finding a successful mapping ($mpng$), the algorithm applies the patch to ast_t where context variables from the patch are replaced by variables in scope using $mpng$. The algorithm applies the edits (line 10) from the patch $gup_{AU\text{-}C}.gedits$ by performing the operation defined in each edit and starting at the location in ast_t identified by $stmt$. If successful, the function APPLY-UPDATE-PATCH returns a new AST (ast'_t) encoding the update of AU and the algorithm considers the patch as being applicable to the target app. For each edit type, the algorithm identifies statements $pstmts$ in ast_t that can be operated on by performing a preorder traversal in ast_t starting at $stmt$. If an edit operation adds a statement at the beginning of $pstmts$, $stmt$ is updated to be the new initial statement. If any of the edit operations fails, the algorithm returns the `null` value to indicate that the update was not successful. Given $pstmts$, the algorithm operates differently based on edit type. For INSERT edits, the algorithm adds the affected statement as at the earliest point across $pstmts$ based on the placement $plcmnt$ identified in Section 7.3.3. For MOVE edits, the algorithm uses the abstraction of as to select a statement in $pstmts$ based on the position provided by $plcmnt$ and move the statement to its new position $plcmnt_2$ always considering $pstmts$. For UPDATE edits, the algorithm uses the abstraction of as to update a statement in $pstmts$ based on $plcmnt$ with the new statement from the edit. For DELETE edits, the algorithm uses the abstraction of as to delete a statement in $pstmts$ based on $plcmnt$.

After successfully computing ast'_t (based on $stmt$, $mpng$, and $gup_{AU\text{-}C}$), the algorithm

updates the code of ast_t with the code of ast'_t in T and validates the updated app T' using differential testing (line 13). If the validation process is successful, the algorithm documents the code changes in the report $ur_{AU\text{-}C}$ (annotating changes as being validated) and returns the report and the updated app as results. In the case the validation process is not successful, the algorithm documents the code changes in $ur_{AU\text{-}C}$ (annotating changes as being applicable) and proceeds by evaluating the next mapping. The algorithm can be adapted to continue after the first validated update to process remaining patches. The algorithm stops at the earliest time to provide a shorter report to developers.

Finally, after processing all API usages in the API-usage report, this phase produces as output an evolved target app and an API-usage update report where validated and applicable updates are documented. These two artifacts are the final output of APPEVOLVE.

7.4 Evaluation

To determine the effectiveness and efficiency of APPEVOLVE, I implemented it in a tool built in Java and Python. I then evaluated the performance of the tool on a set of real-world apps, using GitHub as the code base where to search for update examples. I also compared the effectiveness of APPEVOLVE with that of LASE [115]. I selected LASE as a baseline because it also distills edit scripts from examples, is the technique most closely related to APPEVOLVE, handles Java programs, and is publicly available [116]. However, it is fair to note that LASE has different goals from APPEVOLVE. In particular, LASE was designed to work with one application at a time, and thus with fairly homogeneous examples. In our context, conversely, examples are collected from different apps and can therefore be quite diverse. In the empirical evaluation, I targeted the following research questions:

- **RQ1:** Can APPEVOLVE successfully update API usages in Android apps?
- **RQ2:** For the update examples identified by APPEVOLVE, how do APPEVOLVE and LASE compare in terms of effectiveness?

- **RQ3:** What is the cost of running APPEVOLVE?

7.4.1 Experimental Benchmarks and Setup

For the empirical evaluation, I used a set of 15 real-world apps from the F-droid catalogue [37]. I chose apps from F-droid as they are categorized by the latest API version that apps support. I used this feature to randomly select three sets of five apps. Each set contains apps whose latest supported API version is the same. The three versions of the API are 22, 23, and 25. I chose these three versions as their following version is a major version release and (at the moment of writing) their usage distribution is above 10% [59]. Additionally, apps in these sets contained at least one different API usage that needs to be updated in the following version of the API according to the documentation of the API [56, 53, 54, 55]. For these three versions, I identified API usages that required update by manually looking at the API documentation and encoded them into three API-change specifications. Encoding API changes into the specifications took us only a few minutes to complete. In addition, the specification needs to be computed only once for each version of the API and can be shared across developers. In future work, I plan to compute the specification automatically using a mix of program analysis and natural language processing techniques. Table 7.1 provides a summary description of the benchmarks considered. Columns *Name*, *Category*, *App Vers*, *API Vers*, and *LOC* report the name, category, version, latest supported API version, and lines of code for a benchmark. To perform the evaluation, I implemented APPEVOLVE in a tool that is built with Java and Python. The tool uses GitHub to search for update examples.

7.4.2 Results

RQ1 (Effectiveness) : To answer RQ1, I applied my technique to the set of 15 benchmarks. Tables 7.2, 7.3, and 7.4 report the results of the evaluation. Overall, APPEVOLVE was able to update 17 out of 20 API usages (success rate of 85%) and 37 out of 41 of their

Table 7.1: Benchmarks used in the evaluation of APPEVOLVE.

ID$_A$	*Name*	*Category*	*App Vers*	*API Vers*	*LOC (K)*
A01	BIPOLALARM	Entertainment	0.1.1	22	4
A02	CONVERSATIONS	Communication	1.8.0	22	53.1
A03	PARKENDD	Navigation	1.2.3	22	18
A04	CLEAN SB	Tools	1.1.4	22	16.4
A05	OPENSUDOKU	Game	2.5.2	22	24.3
A06	WIGLE WIFI	Tools	2.10.0	23	35.7
A07	FOOTGUY	Lifestyle	1.5.0	23	3.4
A08	CALENDAR IE	Productivity	2.4.0	23	8.2
A09	DIOLINUX	News	2.2.2	23	13
A10	SOLAR COMPASS	Navigation	1.0.0	23	14.4
A11	SYMPHONY	Entertainment	1.1.9	25	15
A12	SYSLOG	Tools	2.1.1	25	27.1
A13	MUZEI	Personalization	2.4.0	25	64.4
A14	NOTES	News	1.0.1	25	25.2
A15	ONETWO	Tools	1.1.6	25	20.7

occurrences across benchmarks.

Table 7.2 shows the results of APPEVOLVE's update examples search phase. Column *Old API Usage* shows an old API usage requiring update in at least one of the benchmarks. I identified these API usages with the first phase of APPEVOLVE. The number of API usages (20) is different from the number of benchmarks (15) as some of the benchmarks required multiple API usages to be updated. Column *API Vers* shows the new version of the API that requires the update of the API usage. Column *Files* shows the number of files containing the keywords computed for the given API usage while column *PFiles* shows the number of files processed by the technique. The two columns differ as I gave a timeout of 24 hours to this phase. Column *UE* represents the number of update examples identified by the search. The technique could automatically find at least one update example for all API usages but U12.

Table 7.3 shows the properties of update examples analyzed by the third phase of APPEVOLVE. Column *PUE* represents the number of processed update examples. The number differs from the one in column *UE* of Table 7.2 as the current implementation of the technique requires to encode update examples as Eclipse projects. (I am working on automating this part of the implementation.) I randomly selected up to five update examples resulting

Table 7.2: Results of the update examples search phase of APPEVOLVE.

ID$_U$	*Old API Usage*	*API Vers*	*Files*	*PFiles*	*UE*	*Time*
U01	[addAction]	23	17139	11862	1	24h00m
U02	[getAllNetworkInfo]	23	6502	6502	15	5h41m
U03	[getCurrentHour]	23	6950	6950	19	9h16m
U04	[getCurrentMinute]	23	6532	6532	18	8h55m
U05	[setCurrentHour]	23	3988	3988	17	5h27m
U06	[setCurrentMinute]	23	2977	2977	17	4h48m
U07	[setTextAppearance]	23	35914	26151	21	24h00m
U08	[addGpsStatusListener]	24	587	587	1	58m
U09	[fromHtml]	24	49070	13597	64	24h00m
U10	[release]	24	25420	25420	2	21h50m
U11	[removeGpsStatusListener]	24	467	467	1	52m
U12	[shouldOverrideUrlLoading]	24	7842	7842	0	12h22m
U13	[startDrag]	24	2804	2804	2	3h3m
U14	[abandonAudioFocus]	26	138	138	5	1h03m
U15	[getDeviceId]	26	21144	21144	7	16h51m
U16	[requestAudioFocus]	26	443	443	4	1h30m
U17	[saveLayer]	26	19532	19532	1	11h02m
U18	[setAudioStreamType]	26	3122	3122	16	4h05m
U19	[vibrate(long)]	26	3018	3018	7	14h42m
U20	[vibrate(long[],int)]	26	2930	2930	3	14h43m

from the search. Columns below the *Edits* header show the minimum (*Min*), maximum (*Max*), and average (*Avg*) number of edits computed by comparing the ASTs from the old and new versions in the examples. Columns below the *Related Edits* header show the minimum (*Min*), maximum (*Max*), and average (*Avg*) number of edits computed through dependency analysis. For 14 out of 19 cases, the average number of related edits is lower than the average number of edits. This result shows that it is not uncommon for developers to perform additional changes that are unrelated to the update. Excluding them from generated patches becomes important. At the same time, related edits still involve multiple statements. This situation highlights the need for having a technique that performs updates automatically. Columns below the *Core Proximity Value* header show the minimum (*Min*), maximum (*Max*), and average (*Avg*) value computed to measure the proximity of a patch to the update core. For 11 out of 19 cases, the average core proximity value is different from its min. and max. This result shows that examples perform the update using different operations.

Table 7.4 details the characteristics of the API-usage update phase of APPEVOLVE to-

Table 7.3: Characteristics of the update examples used in the evaluation of APPEVOLVE.

ID_U	*PUE*	*Edits*			*Related Edits*			*Core Proximity Value*			*Time*
		Min	*Max*	*Avg*	*Min*	*Max*	*Avg*	*Min*	*Max*	*Avg*	
U01	1	6	6	6	6	6	6	1	1	1	52s
U02	5	16	22	19	16	21	18.8	0.43	0.56	0.48	14s
U03	5	6	17	9.8	5	16	8.8	0.31	1	0.68	14s
U04	5	7	13	9.2	5	13	8.2	0.38	1	0.7	15s
U05	5	7	19	11.4	5	5	5	1	1	1	15s
U06	5	7	19	9.4	5	5	5	1	1	1	15s
U07	5	5	13	8	5	6	5.2	0.83	1	0.97	15s
U08	1	5	5	5	5	5	5	1	1	1	3s
U09	5	5	14	8.6	5	13	8.2	0.31	0.8	0.54	14s
U10	2	5	26	15.5	5	5	5	1	1	1	4s
U11	1	5	5	5	5	5	5	1	1	1	11s
U12	-	-	-	-	-	-	-	-	-	-	-
U13	2	5	5	5	5	5	5	1	1	1	8s
U14	5	5	17	8.8	5	8	6.2	0.63	1	0.85	26s
U15	5	7	17	12.4	7	17	11.6	0.24	0.57	0.5	26s
U16	4	6	18	11.5	6	15	10.5	0.2	0.5	0.43	26s
U17	1	5	5	5	5	5	5	1	1	1	30s
U18	5	8	11	9	8	10	8.8	0.6	0.75	0.69	29s
U19	5	7	47	22.6	7	9	7.6	0.56	0.71	0.67	25s
U20	3	6	47	33.33	6	13	10.33	0.46	1	0.65	29s

gether with the details on the its validation process. This table illustrates the core results for APPEVOLVE, as it shows the updates that the technique could successfully generate. Column *Appl GUP* shows the number of applicable patches for a given occurrence of an API usage. In four cases APPEVOLVE was not able to identify an applicable patch. For U12 in A09, the technique did not have any update examples for the task. For two occurrences of U11 and one of U08 in A06, available patches required a context variable which was not in scope for the methods containing the usages in the benchmark. Using an interprocedural analysis, APPEVOLVE could identify if such variables are defined in the update. Columns *Edits*, *REdits*, and *CPV* show the properties of validated patches. All validated patches were the first applicable patches that APPEVOLVE generated. In three cases (two of U14 in A11 and U15 in A12) the generated patch does not have maximum core proximity value. In these cases, the patch contains additional statements that reduce the number of context variables needed to use the patch, making the patch applicable over others with higher core proximity value. This situation shows that only selecting statements from the core of the update would prevent updating certain API usages. Columns *SVars* and *CVars*

Table 7.4: Details on validate patches for all API-usage occurrences.

ID$_A$	*ID$_U$*	*Appl GUP*	*Edits*	*REdits*	*CPV*	*SVars*	*CVars*	*Validated*		*Time*
								Auto	*Man*	
A01	U01	1	6	6	1	8	5	✗	✓	1s052ms
	U01	1	6	6	1	8	5	✗	✓	899ms
	U01	1	6	6	1	8	5	✗	✓	783ms
A02	U02	2	16	16	0.56	3	1	✓	✓	2s690ms
	U03	2	7	5	1	6	2	✗	✓	3s964ms
	U04	2	7	5	1	4	2	✗	✓	4s396ms
	U05	5	17	5	1	5	2	✓	✓	5s964ms
	U06	5	7	5	1	7	2	✓	✓	4s760ms
A03	U03	2	7	5	1	12	2	✓	✓	3s186ms
A04	U03	2	7	5	1	2	2	✓	✓	1s558ms
	U04	2	7	5	1	3	2	✓	✓	2s323ms
	U05	5	17	5	1	3	2	✓	✓	1s964ms
	U06	5	7	5	1	4	2	✓	✓	1s636ms
A05	U07	4	11	5	1	16	3	✓	✓	5s278ms
	U07	4	11	5	1	16	3	✓	✓	5s403ms
A06	U08	-	-	-	-	-	-	-	-	3s195ms
	U09	3	5	5	0.8	31	2	✗	✓	9s802ms
	U11	-	-	-	-	-	-	-	-	2s616ms
	U11	-	-	-	-	-	-	-	-	3s414ms
A07	U09	3	5	5	0.8	9	2	✓	✓	809ms
A08	U09	1	5	5	0.8	5	2	✓	✓	1s595ms
	U09	3	5	5	0.8	4	2	✗	✓	1s311ms
	U10	2	26	5	1	3	1	✓	✓	524ms
A09	U12	-	-	-	-	-	-	-	-	-
A10	U13	2	5	5	1	8	4	✗	✓	1s630ms
A11	U14	2	8	8	0.63	8	2	✓	✓	1s060ms
	U14	2	8	8	0.63	7	2	✓	✓	1s030ms
	U16	1	15	15	0.2	11	5	✓	✓	724ms
	U18	3	8	8	0.75	3	1	✓	✓	710ms
A12	U15	1	7	7	0.57	8	2	✓	✓	1s479ms
A13	U17	1	5	5	1	14	7	✓	✓	1s520ms
A14	U18	3	8	8	0.75	2	1	✗	✓	1s211ms
A15	U19	3	47	7	0.71	3	2	✗	✓	1s897ms
	U19	3	47	7	0.71	4	2	✓	✓	1s815ms
	U19	3	47	7	0.71	3	2	✓	✓	825ms
	U19	3	47	7	0.71	3	2	✓	✓	1s066ms
	U19	3	47	7	0.71	10	2	✓	✓	770ms
	U19	3	47	7	0.71	11	2	✓	✓	1s054ms
	U20	1	6	6	1	6	3	✓	✓	927ms
	U20	1	6	6	1	5	3	✗	✓	770ms
	U20	1	6	6	1	12	3	✗	✓	560ms

report the number of scope and context variables in the update.

The columns below the header *Validate* show whether the update was automatically validated (column *Auto* with ✓) and manually validated (column *Man* with ✓). The benchmarks did not have an associated test suite with them. Therefore, the technique generated the test suite using random input generation and automatically validated the patches using differential testing as described in Section 7.3.4. APPEVOLVE could automatically validate 25 patches out of the 37 generated. The technique could not validate 12 patches as

the test suite did not cover the code of the update. I manually validated patches by exercising updates in the benchmarks. I also manually analyzed the code of the update and confirmed that it was following the changes described in the documentation of the API. I could manually validate all 37 patches. I would like to point out that for some instances of the same API usage (*e.g.,* three instances of U01 in A01), the update task was performed on methods with a similar set of variables in scope.

In summary, I believe that the results from this section provide initial evidence that APPEVOLVE can automatically update a large number (85%) of API usages for Android apps. The technique is also able to validate a good number of updates (68%).

RQ2 (Effectiveness Comparison) : To answer RQ2, I applied LASE to the same set of 15 benchmarks considered for RQ1. As inputs to LASE, I used the update examples in Table 7.3, that is, the examples automatically retrieved by APPEVOLVE and used to perform the update tasks in Table 7.4. Unfortunately, LASE was unable to (fully) perform the update tasks considered, for different reasons. For nine tasks, LASE could not identify where to apply the edit script in the target app (Reason #1). For another eight tasks, the generated edit script was incomplete (Reason #2). For one task, LASE applied the edit script at the wrong program location (Reason #3). The remaining update tasks failed for multiple reasons: 10 tasks for #1+#2 and 12 tasks for #2+#3. (In one case, APPEVOLVE did not find update examples.)

After further analyzing the results, I believe that the main reason for LASE's performance is the fact that the tool was designed to operate in a different context. As I mentioned in Section 7.4, LASE was designed to work with update examples extracted from a *single code base*. When presented with the update examples automatically extracted by APPEVOLVE from multiple code bases, LASE had problems handling the diversity in terms of (1) operations used to perform the updates and (2) locations where the updates are performed. In fact, these possible limitations are mentioned in the LASE paper itself [115].

Based on the above results and analysis, I conclude that APPEVOLVE is more effective than LASE when used to automatically perform app updates for API changes.

RQ3 (Efficiency) : To answer RQ3, I measured the time taken by each phase of the technique to process the benchmarks (running on a workstation with 64GB of RAM, one Intel Xeon i7-6700K Skylake 4.0GHz, and Ubuntu 16.04).

Columns labeled with *Time* in Tables 7.2, 7.3, and 7.4 report the time taken by the update examples search, update examples analysis, and API-usage update phases of APPEVOLVE, respectively. The API-usage analysis phase took 28 seconds on average. The time shown in Table 7.4 does not include the time to validate a patch. In the experiments of this evaluation I did not have a test suite available and I automatically generated one using random input generation. I timed out input generation after 10 minutes as related work [147] found that a set of dynamic input generation tools for Android apps reached their maximum coverage within 10 minutes. For this reason, the validation process for each applicable patch takes 20 minutes (10 on the old API and 10 on the new API).

As it is possible to see from Table 7.2, the update examples search phase dominates the other phases as it takes 10 hours 27 minutes on average to complete (reaching a timeout of 24 hours in three cases). The most expensive operation in this phase corresponds to transferring a remote code base to the location on which the analysis is running. Indexing repositories offline could speed-up this phase, which could also be parallelized (and the results shared across developers).

Based on these results, I conclude that the cost of running APPEVOLVE is dominated by its search for update examples, which on average can run overnight when looking for updates in parallel.

7.4.3 Limitations and Threats to Validity

My technique analyzes updates within a method's boundary. For this reason, the technique does not currently handle updates that span across multiple methods. This situation only appeared for two of the 20 cases in Table 7.3. I leave this improvement as future work. The technique is able to automatically identify update examples using keywords computed from the signatures of methods in a new API usage. The technique can not automatically search for examples if there are no methods in the new API usage (*i.e.,* the API usage was removed). I believe that this situation would not generally appear as APIs generally follow a *deprecate-replace-remove* cycle [95]. Despite these limitations, this empirical evaluation shows initial evidence that my technique can be practical and effective.

As it is the case for most empirical evaluation, there are both external and construct threats to validity associated with these results. In terms of external validity, these results might not generalize to other apps or API usages as I considered 20 API usages in 15 benchmarks from 3 versions of the API. To mitigate this threat, I used randomly selected real-world apps from different categories and considered different versions of the API. In terms of construct validity, there might be errors in the implementation of the technique. To mitigate this threat, I extensively inspected the results of the evaluation manually.

CHAPTER 8
RELATED WORK

The work described in this dissertation relates to four main areas of research: (i) test case generation using record and replay, (ii) analysis of inconsistent software behavior, (iii) code synthesis from natural language descriptions, (iv) and example-based program update.

8.1 Test Case Generation Using Record and Replay

Related work explored the idea of using record and replay to automate testing of mobile apps [84, 50, 101, 96, 97, 143, 46, 76, 184]. TESTDROID RECORDER (TR) [84] is a tool, implemented as an Eclipse plugin, that records interactions from a connected device running the application under test (AUT). BARISTA is similar to TR in that they both record interactions at the application layer, however the approach used by TR presents some limitations. First, TR uses identifiers that do not reliably identify elements in the UI. Second, generated tests rely on `sleep` commands, which make tests slow and unreliable. Third, the tool does not suitably instrument the UI of the AUT to process user inputs leading to missed interactions in recorded tests.

ESPRESSO TEST RECORDER (ETR) [84] is part of Android Studio and generates tests by recording user interactions from a connected device running the AUT. Similarly to BARISTA, the tool generates Espresso [49] tests. However, the tool support for defining oracles is limited and tests use references to UI elements that tend to be inaccurate.

ACRT [101] is a research tool that, similarly to TR, generates ROBOTIUM tests from user interactions. ACRT is based on an app instrumentation approach that modifies the layout of the AUT to record user interactions and adds a custom gesture to certain element of the UI to allow the user to define oracles. The support for interactions and oracles is limited and the technique does not consider how to uniquely identify elements in the UI.

SPAG [96] uses SIKULI [180] and ANDROID SCREENCAST [9] to create a system in which the screen of the AUT is redirected to a PC and the user interacts with AUT using the PC. SPAG−C [97] extends SPAG using image comparison methods to validate recorded oracles. The approach for oracle definition presented in SPAG and SPAG−C is minimally invasive, as it does not modify the AUT. However, expressing oracles for a specific element in the UI is a practical challenge and the image comparison approach can miss small but significant differences. MOBIPLAY [143] is a record and replay technique based on remote execution. The technique is similar to BARISTA in that inputs to the AUT are collected at the application layer. However, MOBIPLAY input collection approach requires modifications in the Android software stack. In addition, MOBIPLAY records inputs based on their screen coordinates, while BARISTA collects them so that they are platform-independent. Finally, MOBIPLAY does not support the definition of oracles.

RERAN [46] records low level system events by leveraging the Android `getevents` utility and generates a replay script for the same device. The low level approach presented by RERAN is effective in recording and replaying complex multi-touch gestures. However, generated scripts are not suitable for replay on different devices because recorded interactions are based on screen coordinates. VALERA [76] redesigns and extends RERAN with a stream-oriented record-and-replay approach. MOSAIC [184] extends RERAN to overcome the device fragmentation problem. The technique abstracts low-level events into an intermediate representation before translating them to a target system. RERAN, VALERA, and MOSAIC are powerful techniques for record and replay. However, they do not support oracle definition, which constitute a fundamental aspect of UI testing.

Related work also focused on using record and replay to automate testing of desktop [2, 78, 161, 142, 109] and web applications [118, 157, 24, 136, 8, 156, 67].

8.2 Analysis of Inconsistent Software Behavior

Related work studied inconsistencies in relation to the fragmentation of the Android ecosystem [70, 75, 82, 93, 135, 102, 174, 183]. Han and colleagues [70] are among the first to study the issues generated by the fragmentation. Their work analyzes bug reports from two popular mobile device vendors and proposes a method for tracking fragmentation. In this line of research, Holzinger and colleagues [75] discuss the challenges involved in developing apps due to the differences in size and display resolution of devices. DIFFDROID helps developers in this task by automatically identifying inconsistencies across devices.

Related work also identifies compatibility issues in Android apps [171]. Wei and colleagues propose a technique based on static analysis that identifies compatibility issues using an API-Context pair model. The issues identified by their technique are different from those identified by DIFFDROID, as they are related to platform API evolution and drivers implementation. This technique could therefore also be combined with DIFFDROID to identify a broader set of issues.

DIFFDROID also relates to the work on inconsistency identification for web applications [150, 117, 148, 33, 149]. Roy Choudhary and colleagues [150] propose a technique that crawls the AUT in different browsers, collects DOM trees and screenshots for web pages, and compares collected trees and images to identify inconsistencies. There are differences between mobile and web applications that prevent this and similar techniques to be straightforwardly applied in the mobile context. This technique, for instance, runs browsers so that the size of their visible area is the same, which is an assumption that cannot be made for mobile apps.

8.3 Code Synthesis from Natural Language Descriptions

The problem of synthesizing code from natural language descriptions was studied in different domains [168, 22, 89, 27, 30, 165, 108, 90, 68, 35, 88, 145, 45]. In the following, I

discuss the work most closely related to YAKUSU. Branavan and colleagues [22] and Lau and colleagues [89] independently used NLP techniques to extract concrete actions from text documents for improving productivity of various tasks (*e.g.,* troubleshooting, learning from tutorials). Branavan and colleagues [22] proposed a reinforcement learning approach for translating tutorials of desktop applications into a list of concrete actions. Their approach uses a learning algorithm, trained with a corpus of text documents, to infer actions. YAKUSU uses dependency parsing to infer actions from bug reports because, in the mobile application domain, a set of documents (*i.e.,* bug reports) is not always available for training. Lau and colleagues [89] proposed an automated approach to translate into test cases documents that describe how to accomplish various tasks on the web. Their work and YAKUSU both translate natural language sentences into concrete actions. However, their approach assumes that instructions are properly segmented (*i.e.,* one action per sentence), while YAKUSU is able to automatically identify segments (referred to as clauses in Chapter 6) in a sentence. In a sense, Branavan and colleagues and Lau and colleagues realized that the problem of interpreting arbitrary hand-written documents, although very challenging in general, becomes more amenable to machine processing when the input language is restricted to a certain domain. YAKUSU builds on this idea, while weakening the assumptions on the input made by the aforementioned techniques.

Thummalapenta and colleagues [165] proposed ATA, an approach to translate web application tests, written in natural language by professional testers, into their corresponding scripts. Both YAKUSU and ATA identify actions using dependency parsing and account for the fact that multiple choices for a target might be present during test generation. However, there are generally considerable differences between professionally written test specifications and bug reports, which prevents ATA from being straightforwardly applied to bug reports. For instance, bug reports written by users do not generally follow a precise structure and can use different levels of abstractions in describing the steps to reproduce an issue. YAKUSU takes these aspects into account by mapping the description of the bug report to

an ontology of the app. Additionally, YAKUSU is also able to handle cases in which some necessary steps of the test are not explicitly mentioned in the bug report.

Le and colleagues [90] proposed SMARTSYNTH, a technique that combines NLP and program synthesis to produce automation scripts. For example, SMARTSYNTH could handle an automation script like the following one: "When I receive a new SMS message, if the phone is connected to my car's bluetooth, it reads out loud the message content and replies the sender 'I'm driving'." SMARTSYNTH and YAKUSU differs in terms of their application context, as SMARTSYNTH generates system events (*e.g.,* "Turn off GPS") as opposed to UI events. In this sense, SMARTSYNTH can be seen as complementary to YAKUSU, which could leverage system events to increase the range of bug reports it can handle (*e.g.,* "Turn off GPS before clicking button x").

More broadly, YAKUSU relates to the area of field failure reproduction (*e.g.,* [161, 129, 13, 80, 81, 185, 130]). I believe that YAKUSU and these alternative techniques tend to have complementary advantages and disadvantages. In particular, techniques for failure reproduction based on partial information (*e.g.,* stack traces, partial event sequences) do not require NLP but tend to involve heavy-duty analysis. Overall, these techniques and YAKUSU are mostly orthogonal, and a developer may decide which one to use for a specific bug report based on the information available.

8.4 Example-Based Program Update

Related work studied example-based program update [7, 131, 114, 115, 146, 123]. LASE [115] performs repetitive edits in a program by computing an edit script from examples. The edit script is computed by selecting common edit operations across examples and uses clone detection, subgraph analysis, and dependency analysis to identify locations where to apply the edit script. By selecting common edit operations, LASE is sensitive to differences in update examples, which can lead to incomplete edit scripts. APPEVOLVE relates to LASE in the way it computes edit operations from a single update example. However,

APPEVOLVE ranks edit scripts based on their edit operations to avoid creating incomplete edit scripts. LASE extends SYDIT [114] to support multiple examples to improve generalization of edit scripts. APPEVOLVE uses multiple examples to prioritize the ones that best capture the core of the update. RASE [113] extends LASE to target clone removal.

REFAZER [146] learns code transformations from examples by leveraging a domain-specific language that describes a space of program transformations that commonly occur in practice. Code transformations are composed by rewrite rules that consist of two parts: a location expression and an operation. The location expression is used to determine applicability of a rewrite rule and it is based on a path expression on the AST computed from examples. Because this location expression is designed to target repetitive edits, it can be difficult to perform updates at structurally different locations. This characteristic is also shared by ARES [36] and VURLE [105]. ARES computes update patterns using an example ordering process, a set of adjustment rules, and determines update applicability based on example locations. VURLE uses update examples and their code location to detect and repair security vulnerabilities. Santos et al. [155] study repetitive edits by looking at three different ways (structural, AST based, and information retrieval) to identify change locations. APPEVOLVE generalizes the location where to apply edit scripts by looking at the variables in scope within methods that contain API usages and uses differential testing to validate updates.

A4 [87] learns API migration patterns from update examples and applies these patterns to the source code of Android apps. APPEVOLVE differs from A4 as it identifies update examples in remote repositories, handles changes in return values, and is able to prioritize examples that are closely related to the core of the update shared across examples. Update examples have also been used by HIREBUILD [73] to generate updates for build scripts.

Other techniques (*e.g.,* [131, 14, 123, 176, 31, 166]) focus on suggesting code edits but do not modify target code. LIBSYNC uses graph based techniques to identify changes between two versions of an API. It then analyzes client applications that migrated to the new

version of the API to identify usage adaptation patterns and suggest edit examples. LIBSYNC does not transform target code due to the inability to abstract suggested edits and developers need to perform this task manually. PARC [14] suggests parameters for method calls based on static type analysis and a history of parameter usage. SYSEDMINER [123] finds unknown systematic edits in a code base using a closed frequent itemset mining algorithm. Thung et. al. [166] propose an approach that recommends code changes for backporting of device drivers.

My work also relates to the area of transferring code across software systems (*e.g.,* [172, 72, 140, 16, 160, 159]). The key idea behind this line of research is to transfer code implementing a desired functionality from one software system to another. This task can be performed with the objective to add new functionality to the system (*e.g.,* [72, 16, 159]), improve its performance (*e.g.,* [140]), and repair existing functionality (*e.g.,* [172, 160]). APPEVOLVE and this line of research share the idea of transferring code between software systems but APPEVOLVE focuses on the software evolution task and uses the version history of a software system to automatically identify the code to be transferred.

More generally, APPEVOLVE relates to program repair techniques (*e.g.,* [12, 138, 66, 86, 85, 112, 104, 175, 177, 103, 139, 163, 79, 77]). These techniques use a variety of search algorithms (*e.g.,* genetic programming) to modify statements in a program and repair it. APPEVOLVE relates to some of these techniques (*e.g.,* [104, 103, 79, 91]) as they also use examples but they do so to improve the effectiveness of their search strategy. Compared to these techniques, APPEVOLVE requires a significantly smaller number of examples.

In relation to my work on API-usage update, the literature [111, 98, 17, 99, 70, 82, 178] studied problems associated with the quickly evolving Android API and with the fragmentation of the ecosystem. The study from McDonnell and colleagues [111] shows that 28% of API references in apps are outdated. These studies led to work on detecting issues caused by the API and the fragmentation [171, 39, 94, 74]. APPEVOLVE extends the technique from He and colleagues [74] to identify API usages that should be updated.

CHAPTER 9
CONCLUSION

Because mobile apps are becoming increasingly prevalent in our daily lives, it is imperative to improve their software quality. Automated testing and maintenance techniques help us in improving software quality, but, unfortunately, the current support provided by these techniques for mobile apps is limited and based on mostly manual, time-consuming approaches. In fact, testing of mobile apps is today mainly performed manually, as app developers report that they lack effective methods and tools to do otherwise. In addition, because apps must run on platforms with different properties, developers are also faced with the problem of checking that the apps' behavior is consistent across platforms, which further aggravates the situation. Because testing can not generally reveal all bugs, developers must also quickly react to reported field failures and performing this task can be extremely time-consuming, especially in the presence of a large number an possibly incomplete reports. Finally, apps rely heavily on the API provided by the underlying operating system, and, because the API is frequently updated, developers need to spend time adapting apps to the new API to avoid issues that the changes introduce.

The overarching goal of this dissertation is to improve software quality before and after release by devising automated techniques that address current research problems and challenges in testing and maintenance of mobile apps. To this end, I devised a family of testing and maintenance techniques: BARISTA, DIFFDROID, YAKUSU, and APPEVOLVE. BARISTA and DIFFDROID are testing techniques. BARISTA generates platform-independent test cases through record and replay. DIFFDROID identifies cross-platform inconsistencies through differential testing and UI modeling. YAKUSU and APPEVOLVE are maintenance techniques. YAKUSU translates natural language bug reports into test cases. APPEVOLVE performs API-usage updates by analyzing how other developers performed

corresponding updates. I showed how my techniques can improve software quality by performing a series of empirical evaluations on real-world apps. The evaluations showed that the techniques are both effective and efficient.

9.1 Future Work

In the future, software will continue to play a major role in our lives and we must continue improving software quality. The work presented in this dissertation addresses research problems and challenges in testing and maintenance of mobile apps but it also provides a solid foundation for future work.

9.1.1 Generating Test Cases

There are multiple opportunities for future research in relation to my work on test case generation through record and reply. One possible direction could investigate how to factor out repetitive action sequences, such as app initialization, so that testers do not have to repeat them for every test. Another research direction could study how to add sandboxing capabilities, so that test cases generated through record and replay are resilient to changes in the environment. A third direction could investigate the use of fuzzing for generating extra tests by augmenting those recorded, possibly driven by specific coverage goals.

9.1.2 Analyzing Cross-Platform Software Behavior

I can see multiple research directions for future work in analyzing cross-platform behavior of apps and software in general. One research direction could study how to identify when an app does not behave as expected due to differing capabilities of hardware components across platforms. Another research direction could explore how to detect inconsistencies that emerge from API customizations. The solutions I envision will compare execution traces from different platforms using different levels of execution abstractions. The definition of these abstractions is a core aspect of the research. A third direction, could investigate

how to change an app's code to account for differences in hardware, API, and display capabilities. To this end, it could be worth investigating solutions that explore a program repair space constrained by a model of the expected behavior and explore approaches that leverage abstractions of how other apps' developers dealt with similar issues. Another research direction could explore inconsistencies between multiple implementations of the same software, e.g., between an Android app and the corresponding iOS app, but also between an app and a corresponding web application.

9.1.3 Analyzing Bug Reports

In this dissertation I have shown how it is possible to translate natural language bug reports into test cases by combining natural language processing techniques with static and dynamic analyses. I believe that we can take advantage of this combination to define additional automated techniques that will help us in better analyze bug reports. One possible research direction for future work in this area is investigating how to help users of apps in writing actionable bug reports. The solution I envision will provide suggestions to users by analyzing sentences being typed, mapping their content to a model of the execution state, and proposing text to users based on actions that are possible in a certain GUI state. This solution will guide users in writing reports free of ambiguity or missing information. A second research direction could focus on investigating how to triage app reviews containing bug descriptions. Specifically, I consider it possible to identify duplicate reviews by extracting and comparing observed and expected behavior from their descriptions. A third research direction for future work could explore how to automatically generate natural language bug reports from users' executions.

9.1.4 Supporting Changes to the Software Environment

I also see multiple research directions for future work in regard to my work on updating API usages and supporting changes to the software environment. One research direction could

investigate how to handle updates that span across multiple methods. A second research direction could study how to combine example-based program update with program repair to account for update examples that are not complete. A third research direction could explore how to automatically compute API change specifications based on a mix of program analysis and natural language processing techniques, starting from the API documentation.

REFERENCES

[1] 0nko, *Notification icon: App crash when publishing a post*, https://github.com/wordpress-mobile/WordPress-Android/issues/5497, 2017.

[2] Abbot, *Abbot java gui test framework*, http://abbot.sourceforge.net/doc/overview.shtml, 2002.

[3] D. Amalfitano, A. R. Fasolino, P. Tramontana, S. De Carmine, and A. M. Memon, "Using gui ripping for automated testing of android applications," in *Proceedings of the IEEE/ACM International Conference on Automated Software Engineering*, 2012.

[4] Amazon, *Device farm*, https://aws.amazon.com/device-farm, 2017.

[5] "Analysis of the errors caused by the fragmentation of the android ecosystem: An empirical study," *Georgia Institute of Technology*, 2019.

[6] S. Anand, M. Naik, M. J. Harrold, and H. Yang, "Automated concolic testing of smartphone apps," in *Proceedings of the 20th ACM SIGSOFT International Symposium on the Foundations of Software Engineering*, 2012.

[7] J. Andersen and J. L. Lawall, "Generic patch inference," in *2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, 2008.

[8] S. Andrica and G. Candea, "Warr: A tool for high-fidelity web application record and replay," in *Proceedings of the International Conference on Dependable Systems and Networks*, 2011.

[9] AndroidScreencast, *Androidscreencast*, https://code.google.com/p/androidscreencast, 2014.

[10] G. Angeli, M. J. J. Premkumar, and C. D. Manning, "Leveraging linguistic structure for open domain information extraction," in *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing*, 2015.

[11] J. Annuzzi Jr, L. Darcey, and S. Conder, *Advanced Android Application Development*. Pearson Education, 2014.

[12] A. Arcuri and X. Yao, "A novel co-evolutionary approach to automatic software bug fixing," in *2008 IEEE Congress on Evolutionary Computation (IEEE World Congress on Computational Intelligence)*, 2008, pp. 162–168.

[13] S. Artzi, S. Kim, and M. D. Ernst, "Recrash: Making software failures reproducible by preserving object states," in *22nd European Conference on Object-Oriented Programming*, 2008.

[14] M. Asaduzzaman, C. K. Roy, S. Monir, and K. A. Schneider, "Exploring api method parameter recommendations," in *2015 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2015, pp. 271–280.

[15] T. Azim and I. Neamtiu, "Targeted and depth-first exploration for systematic testing of android apps," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications*, 2013.

[16] E. T. Barr, M. Harman, Y. Jia, A. Marginean, and J. Petke, "Automated software transplantation," in *Proceedings of the 2015 International Symposium on Software Testing and Analysis*, Baltimore, MD, USA: ACM, 2015, pp. 257–269.

[17] G. Bavota, M. Linares-Vásquez, C. E. Bernal-Cardenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "The impact of api change- and fault-proneness on the user ratings of android apps," *IEEE Transactions on Software Engineering*, 2015.

[18] N. Bettenburg, S. Just, A. Schröter, C. Weiss, R. Premraj, and T. Zimmermann, "What makes a good bug report?" In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2008.

[19] P. Bhattacharya, L. Ulanova, I. Neamtiu, and S. C. Koduru, "An empirical analysis of bug reports and bug fixing in open source android apps," in *17th European Conference on Software Maintenance and Reengineering*, 2013.

[20] I. Blair, *Mobile app download and usage statistics (2019)*, `https://buildfire.com/app-statistics`, 2019.

[21] D. Bolton, *88% of people will abandon an app because of bugs*, `https://www.applause.com/blog/app-abandonment-bug-testing`, 2017.

[22] S. R. K. Branavan, H. Chen, L. S. Zettlemoyer, and R. Barzilay, "Reinforcement learning for mapping instructions to actions," in *Proceedings of the 47th Annual Meeting of the Association for Computational Linguistics and the 4th International Joint Conference on Natural Language Processing of the AFNLP*, 2009.

[23] Bullnados, *Main and nightly version crashing all time on lg g4*, https://github.com/nextcloud/android/issues/760, 2017.

[24] B. Burg, R. Bailey, A. J. Ko, and M. D. Ernst, "Interactive record/replay for web application debugging," in *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, 2013.

[25] Capgemini, *World quality report 2017-18*, https://www.capgemini.com/service/world-quality-report-2017-18, 2018.

[26] J. Carey, *Whatsapp may have a huge problem and fans are not happy*, https://www.express.co.uk/life-style/science-technology/1072506/WhatsApp-bug-error-chat-history-deleted, 2019.

[27] C. S. Carlos, "Natural language programming using class sequential rules," in *Proceedings of the Fifth International Joint Conference on Natural Language Processing*, 2011.

[28] W. Choi, G. Necula, and K. Sen, "Guided gui testing of android apps with minimal restart and approximate learning," in *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages, and Applications*, 2013.

[29] P. Costa, A. C. R. Paiva, and M. Nabuco, "Pattern based gui testing for mobile applications," in *2014 9th International Conference on the Quality of Information and Communications Technology*, 2014.

[30] A. Cozzie, M. Finnicum, and S. T. King, "Macho: Programming with man pages," in *13th Workshop on Hot Topics in Operating Systems*, 2011.

[31] B. Dagenais and M. P. Robillard, "Recommending adaptive changes for framework evolution," in *Proceedings of the 30th International Conference on Software Engineering*, ser. ICSE '08, New York, NY, USA: ACM, 2008.

[32] DailyDozen, *Dailydozen*, https://play.google.com/store/apps/details?id=org.nutritionfacts.dailydozen, 2017.

[33] V. Dallmeier, M. Burger, T. Orth, and A. Zeller, "Webmate: A tool for testing web 2.0 applications," in *Proceedings of the Workshop on JavaScript Tools*, 2012.

[34] U. Dependencies, *Universal dependencies*, http://universaldependencies.org, 2018.

[35] A. Desai, S. Gulwani, V. Hingorani, N. Jain, A. Karkare, M. Marron, S. R., and S. Roy, "Program synthesis using natural language," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[36] G. Dotzler, M. Kamp, P. Kreutzer, and M. Philippsen, "More accurate recommendations for method-level changes," in *Proceedings of the 2017 11th joint meeting on foundations of software engineering*, New York, NY, USA: ACM, 2017, pp. 798–808.

[37] *F-droid*, `https://f-droid.org`, Oct. 2018.

[38] M. Fazzini, E. N.D. A. Freitas, S. R. Choudhary, and A. Orso, "Barista: A technique for recording, encoding, and running platform independent android tests," in *Proceedings of the 2017 IEEE International Conference on Software Testing, Verification and Validation*, 2017.

[39] M. Fazzini and A. Orso, "Automated cross-platform inconsistency detection for mobile apps," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[40] M. Fazzini, M. Prammer, M. d'Amorim, and A. Orso, "Automatically translating bug reports into test cases for mobile apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[41] M. Fazzini, Q. Xin, and A. Orso, "Automated api-usage update for android apps," in *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2019.

[42] B. Fluri, M. Wuersch, M. Pinzger, and H. Gall, "Change distilling: Tree differencing for fine-grained source code change extraction," *IEEE Transactions on software engineering*, 2007.

[43] GitHub, *Github*, `https://github.com`, 2018.

[44] ——, *Manually creating a single issue template for your repository*, `https://help.github.com/articles/manually-creating-a-single-issue-template-for-your-repository`, 2018.

[45] A. Goffi, A. Gorla, M. D. Ernst, and M. Pezzè, "Automatic generation of oracles for exceptional behaviors," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016.

[46] L. Gomez, I. Neamtiu, T. Azim, and T. Millstein, "Reran: Timing- and touch-sensitive record and replay for android," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[47] Google, *Google play*, https://play.google.com/store, 2008.

[48] ——, *Automating user interface tests*, https://developer.android.com/training/testing/ui-testing, 2013.

[49] ——, *Espresso*, https://developer.android.com/training/testing/espresso, 2013.

[50] ——, *Espresso test recorder*, https://developer.android.com/studio/test/espresso-test-recorder, 2016.

[51] ——, *Monkey*, https://developer.android.com/studio/test/monkey.html, 2017.

[52] ——, *Ui overview*, https://developer.android.com/guide/topics/ui/declaring-layout, 2017.

[53] ——, *Android api differences report (v23)*, https://developer.android.com/sdk/api_diff/23/changes, 2018.

[54] ——, *Android api differences report (v24)*, https://developer.android.com/sdk/api_diff/24/changes, 2018.

[55] ——, *Android api differences report (v26)*, https://developer.android.com/sdk/api_diff/26/changes, 2018.

[56] ——, *Api reference*, https://developer.android.com/reference, 2018.

[57] ——, *Connectivitymanager*, https://developer.android.com/reference/android/net/ConnectivityManager.html#getAllNetworkInfo(), 2018.

[58] ——, *Custom view components*, https://developer.android.com/training/custom-views/index.html, 2018.

[59] ——, *Distribution dashboard*, https://developer.android.com/about/dashboards, 2018.

[60] ——, *Google news vectors negative 300*, https://drive.google.com/file/d/0B7XkCwpI5KDYNlNUTTlSS21pQmM, 2018.

[61] ——, *Reporting bugs*, https://source.android.com/setup/report-bugs, 2018.

[62] ——, *Android*, https://www.android.com, 2019.

[63] ——, *Application fundamentals*, https://developer.android.com/guide/components/fundamentals?hl=en, 2019.

[64] ——, *Compatibility test suite*, https://source.android.com/compatibility/cts, 2019.

[65] ——, *Platform architecture*, https://developer.android.com/guide/platform, 2019.

[66] C. L. Goues, M. Dewey-Vogt, S. Forrest, and W. Weimer, "A systematic study of automated program repair: Fixing 55 out of 105 bugs for $8 each," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[67] M. Grechanik, Q. Xie, and C. Fu, "Creating gui testing tools using accessibility technologies," in *Proceedings of the 2009 IEEE International Conference on Software Testing, Verification, and Validation Workshops*, 2009.

[68] T. Gvero and V. Kuncak, "Synthesizing java expressions from free-form queries," in *Proceedings of the 2015 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2015.

[69] K. Hakata and H. Imai, "Algorithms for the longest common subsequence problem for multiple strings based on geometric maxima," *Optimization Methods and Software*, 1998.

[70] D. Han, C. Zhang, X. Fan, A. Hindle, K. Wong, and E. Stroulia, "Understanding android fragmentation with topic analysis of vendor-specific bugs," in *Proceedings of the 2012 Working Conference on Reverse Engineering*, 2012.

[71] S. Hao, B. Liu, S. Nath, W. G. Halfond, and R. Govindan, "Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps," in *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, 2014.

[72] M. Harman, W. B. Langdon, and W. Weimer, "Genetic programming for reverse engineering," in *2013 20th Working Conference on Reverse Engineering (WCRE)*, 2013, pp. 1–10.

[73] F. Hassan and X. Wang, "Hirebuild: An automatic approach to history-driven repair of build scripts," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, New York, NY, USA: ACM, 2018, pp. 1078–1089.

[74] D. He, L. Li, L. Wang, H. Zheng, G. Li, and J. Xue, "Understanding and detecting evolution-induced compatibility issues in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018.

[75] A. Holzinger, P. Treitler, and W. Slany, "Making apps useable on multiple different mobile platforms: On interoperability for business application development on smartphones," in *International Conference on Availability, Reliability, and Security*, 2012.

[76] Y. Hu, T. Azim, and I. Neamtiu, "Versatile yet lightweight record-and-replay for android," in *Proceedings of the 2015 International Conference on Object Oriented Programming Systems Languages & Applications*, 2015.

[77] J. Hua, M. Zhang, K. Wang, and S. Khurshid, "Towards practical program repair with on-demand candidate generation," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, New York, NY, USA: ACM, 2018, pp. 12–23.

[78] Jacareto, *Jacareto*, `http://sourceforge.net/projects/jacareto`, 2005.

[79] J. Jiang, Y. Xiong, H. Zhang, Q. Gao, and X. Chen, "Shaping program repair space with existing patches and similar code," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[80] W. Jin and A. Orso, "Bugredux: Reproducing field failures for in-house debugging," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[81] ——, "F3: Fault localization for field failures," in *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, 2013.

[82] M. E. Joorabchi, A. Mesbah, and P. Kruchten, "Real challenges in mobile app development," in *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, 2013.

[83] D. Jurafsky and J. H. Martin, *Speech and language processing*. Pearson Education, 2014.

[84] J. Kaasila, D. Ferreira, V. Kostakos, and T. Ojala, "Testdroid: Automated remote ui testing on android," in *Proceedings of the 11th International Conference on Mobile and Ubiquitous Multimedia*, 2012.

[85] Y. Ke, K. T. Stolee, C. Le Goues, and Y. Brun, "Repairing programs with semantic code search (T)," in *Proceedings of 30th IEEE/ACM International Conference on Automated Software Engineering*, 2015.

[86] D. Kim, J. Nam, J. Song, and S. Kim, "Automatic patch generation learned from human-written patches," in *2013 35th International Conference on Software Engineering (ICSE)*, 2013.

[87] M. Lamothe, W. Shang, and T. Chen, "A4: automatically assisting android API migrations using code examples," *CoRR*, 2018.

[88] M. Landhäußer, S. Weigelt, and W. F. Tichy, "Nlci: A natural language command interpreter," *Automated Software Engineering*, 2017.

[89] T. A. Lau, C. Drews, and J. Nichols, "Interpreting written how-to instructions," in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, 2009.

[90] V. Le, S. Gulwani, and Z. Su, "Smartsynth: Synthesizing smartphone automation scripts from natural language," in *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, 2013.

[91] X.-B. D. Le, D. Lo, and C. Le Goues, "History driven program repair," in *Proceedings of the IEEE 23rd International Conference on Software Analysis, Evolution and Reengineering*, Osaka, Japan: IEEE, 2016, pp. 213–224.

[92] D. Lee, *How to write a bug report that will make your engineers love you*, `https://testlio.com/blog/the-ideal-bug-report`, 2016.

[93] H. Li, X. Lu, X. Liu, T. Xie, K. Bian, F. X. Lin, Q. Mei, and F. Feng, "Characterizing smartphone usage patterns from millions of android users," in *Proceedings of the 2015 ACM Conference on Internet Measurement Conference*, 2015.

[94] L. Li, T. F. Bissyandé, H. Wang, and J. Klein, "Cid: Automating the detection of api-related compatibility issues in android apps," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2018.

[95] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated android apis," in *Proceedings of the 15th International Conference on Mining Software Repositories*, New York, NY, USA: ACM, 2018, pp. 254–264.

[96] Y.-D. Lin, E.-H. Chu, S.-C. Yu, and Y.-C. Lai, "Improving the accuracy of automated gui testing for embedded systems," *Software, IEEE*, 2014.

[97] Y.-D. Lin, J. Rojas, E.-H. Chu, and Y.-C. Lai, "On the accuracy, efficiency, and reusability of automated test oracles for android devices," *Software Engineering, IEEE Transactions on*, 2014.

[98] M. Linares-Vásquez, G. Bavota, C. Bernal-Cárdenas, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "Api change and fault proneness: A threat to the success of android apps," in *Proceedings of the 9th joint meeting on foundations of software engineering*, 2013.

[99] M. Linares-Vásquez, G. Bavota, M. Di Penta, R. Oliveto, and D. Poshyvanyk, "How do api changes trigger stack overflow discussions? a study on the android sdk," in *Proceedings of the 22nd International Conference on Program Comprehension*, 2014.

[100] M. Linares-Vásquez, C. Bernal-Cardenas, K. Moran, and D. Poshyvanyk, "How do developers test android applications?" In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2017, pp. 613–622.

[101] C. H. Liu, C. Y. Lu, S. J. Cheng, K. Y. Chang, Y. C. Hsiao, and W. M. Chu, "Capture-replay testing for android applications," in *Computer, Consumer and Control (IS3C), 2014 International Symposium on*, 2014.

[102] Y. Liu, C. Xu, and S.-C. Cheung, "Characterizing and detecting performance bugs for smartphone applications," in *Proceedings of the 36th International Conference on Software Engineering*, 2014.

[103] F. Long, P. Amidon, and M. Rinard, "Automatic inference of code transforms for patch generation," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, Paderborn, Germany: ACM, 2017, pp. 727–739.

[104] F. Long and M. Rinard, "Automatic patch generation by learning correct code," in *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2016.

[105] S. Ma, F. Thung, D. Lo, C. Sun, and R. H. Deng, "Vurle: Automatic vulnerability detection and repair by learning from examples," S. N. Foley, D. Gollmann, and E. Snekkenes, Eds., Cham: Springer International Publishing, 2017, pp. 229–246.

[106] A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: An input generation system for android apps," in *Proceedings of the 2013 Joint Meeting on Foundations of Software Engineering*, 2013.

[107] R. Mahmood, N. Mirzaei, and S. Malek, "Evodroid: Segmented evolutionary testing of android apps," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, 2014.

[108] M. H. Manshadi, D. Gildea, and J. F. Allen, "Integrating programming by example and natural language programming," in *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, 2013.

[109] MarathonITE, *Marathonite powerful tools for creating resilient test suites*, `http://marathontesting.com`, 2008.

[110] M. de Marneffe, B. MacCartney, and C. D. Manning, "Generating typed dependency parses from phrase structure parses," in *Proceedings of the Fifth International Conference on Language Resources and Evaluation*, 2006.

[111] T. McDonnell, B. Ray, and M. Kim, "An empirical study of api stability and adoption in the android ecosystem," in *Proceedings of the 29th IEEE International Conference on Software Maintenance*, 2013.

[112] S. Mechtaev, J. Yi, and A. Roychoudhury, "Angelix: Scalable multiline program patch synthesis via symbolic analysis," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[113] N. Meng, L. Hua, M. Kim, and K. S. McKinley, "Does automated refactoring obviate systematic editing?" In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ser. ICSE '15, Piscataway, NJ, USA: IEEE Press, 2015, pp. 392–402.

[114] N. Meng, M. Kim, and K. S. Mckinley, "Systematic editing: Generating program transformations from an example," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming Language Design and Implementation*, 2011.

[115] N. Meng, M. Kim, and K. S. McKinley, "Lase: Locating and applying systematic edits by learning from examples," in *Proceedings of the 35th International Conference on Software Engineering*, 2013.

[116] ——, *Lase tool*, `http://people.cs.vt.edu/nm8247/research.html`, Jan. 2013.

[117] A. Mesbah and M. R. Prasad, "Automated cross-browser compatibility testing," in *Proceedings of the 33rd International Conference on Software Engineering*, 2011.

[118] J. Mickens, J. Elson, and J. Howell, "Mugshot: Deterministic capture and replay for javascript applications," in *Proceedings of the 7th USENIX Symposium on Networked Systems Design and Implementation*, 2010.

[119] T. Mikolov, K. Chen, G. Corrado, and J. Dean, "Efficient estimation of word representations in vector space," *CoRR*, 2013.

[120] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean, "Distributed representations of words and phrases and their compositionality," in *Advances in Neural Information Processing Systems 26: 27th Annual Conference on Neural Information Processing Systems 2013*, 2013.

[121] W. Miller, Eugene, and W. Myers, "A file comparison program," *Software: Practice and Experience*, 1985.

[122] G. Miner, J. Elder, T. Hill, R. Nisbet, D. Delen, and A. Fast, *Practical Text Mining and Statistical Analysis for Non-structured Text Data Applications*. Academic Press, 2012.

[123] T. Molderez, R. Stevens, and C. De Roover, "Mining change histories for unknown systematic edits," in *Proceedings of the 14th International Conference on Mining Software Repositories*, 2017.

[124] C. Moody, *A word is worth a thousand vectors*, `https://multithreaded.stitchfix.com/blog/2015/03/11/word-is-worth-a-thousand-vectors`, 2015.

[125] R. M.L. M. Moreira, A. C. R. Paiva, and A. Memon, "A pattern-based approach for GUI modeling and testing," in *2013 IEEE 24th International Symposium on Software Reliability Engineering (ISSRE)*, 2013.

[126] I. C. Morgado and A. C. R. Paiva, "Testing approach for mobile applications through reverse engineering of ui patterns," in *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, 2015.

[127] J. Munkres, "Algorithms for the assignment and transportation problems," *Journal of the Society of Industrial and Applied Mathematics*, 1957.

[128] K. G. Murty, "Letter to the editor - an algorithm for ranking all the assignments in order of increasing cost," *Operations Research*, 1968.

[129] S. Narayanasamy, G. Pokam, and B. Calder, "Bugnet: Continuously recording program execution for deterministic replay debugging," in *32nd International Symposium on Computer Architecture*, 2005.

[130] M. Nayrolles, A. Hamou-Lhadj, S. Tahar, and A. Larsson, "Jcharming: A bug reproduction approach using crash traces and directed model checking," in *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, 2015.

[131] H. A. Nguyen, T. T. Nguyen, G. Wilson Jr., A. T. Nguyen, M. Kim, and T. N. Nguyen, "A graph-based approach to api usage adaptation," in *Proceedings of the*

ACM International Conference on Object Oriented Programming Systems Languages and Applications, 2010.

[132] OpenSignal, *Android fragmentation*, https://opensignal.com/reports/2015/08/android-fragmentation, 2015.

[133] J. Ostrander, *Android UI Fundamentals: Develop and Design*. Peachpit Press, 2012.

[134] C. Patel, A. Patel, and D. Patel, "Optical character recognition by open source ocr tool tesseract: A case study," *International Journal of Computer Applications*, 2012.

[135] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: A first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks*, 2011.

[136] K. Pattabiraman and B. Zorn, "Dodom: Leveraging dom invariants for web 2.0 application robustness testing," in *Proceedings of the IEEE 21st International Symposium on Software Reliability Engineering*, 2010.

[137] R. Pear, *Sebelius apologizes for health site's malfunctions*, https://www.nytimes.com/2013/10/31/us/politics/sebelius-apologizes-for-health-sites-malfunctions.html, 2013.

[138] J. H. Perkins, S. Kim, S. Larsen, S. Amarasinghe, J. Bachrach, M. Carbin, C. Pacheco, F. Sherwood, S. Sidiroglou, G. Sullivan, W.-F. Wong, Y. Zibin, M. D. Ernst, and M. Rinard, "Automatically patching errors in deployed software," in *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles*, 2009.

[139] J. Petke, S. O. Haraldsson, M. Harman, W. B. Langdon, D. R. White, and J. R. Woodward, "Genetic improvement of software: A comprehensive survey," *IEEE Transactions on Evolutionary Computation*, pp. 415–432, 2018.

[140] J. Petke, M. Harman, W. B. Langdon, and W. Weimer, "Using genetic improvement and code transplants to specialise a c++ program to a problem class," in *Genetic Programming*, M. Nicolau, K. Krawiec, M. I. Heywood, M. Castelli, P. García-Sánchez, J. J. Merelo, V. M. Rivas Santos, and K. Sim, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2014, pp. 137–149.

[141] H. Pettit, *Bizarre whatsapp bug is 'deleting old messages' and people are panicking*, https://www.thesun.co.uk/tech/8198167/whatsapp-bizarre-bug-deletes-old-messages, 2019.

[142] Pounder, *Pounder*, https://sourceforge.net/projects/pounder, 2002.

[143] Z. Qin, Y. Tang, E. Novak, and Q. Li, "Mobiplay: A remote execution based record-and-replay tool for mobile applications," in *Proceedings of the 2016 International Conference on Software Engineering*, 2016.

[144] J. R. Quinlan, *C4. 5: programs for machine learning*. Elsevier, 2014.

[145] M. Raghothaman, Y. Wei, and Y. Hamadi, "Swim: Synthesizing what i mean: Code search and idiomatic snippet synthesis," in *Proceedings of the 38th International Conference on Software Engineering*, 2016.

[146] R. Rolim, G. Soares, L. D'Antoni, O. Polozov, S. Gulwani, R. Gheyi, R. Suzuki, and B. Hartmann, "Learning syntactic program transformations from examples," in *Proceedings of the 39th International Conference on Software Engineering*, 2017.

[147] S. Roy Choudhary, A. Gorla, and A. Orso, "Automated test input generation for android: Are we there yet? (e)," in *Proceedings of the 2015 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2015.

[148] S. Roy Choudhary, M. R. Prasad, and A. Orso, "Crosscheck: Combining crawling and differencing to better detect cross-browser incompatibilities in web applications," in *2012 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[149] ——, "X-pert: Accurate identification of cross-browser issues in web applications," in *Proceedings of the 2013 International Conference on Software Engineering*, 2013.

[150] S. Roy Choudhary, H. Versee, and A. Orso, "Webdiff: Automated identification of cross-browser issues in web applications," in *2010 IEEE International Conference on Software Maintenance (ICSM)*, 2010.

[151] Y. Rubner, C. Tomasi, and L. J. Guibas, "The earth mover's distance as a metric for image retrieval," *International journal of computer vision*, 2000.

[152] C. Sacramento and A. C. R. Paiva, "Web application model generation through reverse engineering and ui pattern inferring," in *2014 9th International Conference on the Quality of Information and Communications Technology*, 2014.

[153] J. Saito, *Making a case for letter case*, https://medium.com/@jsaito/making-a-case-for-letter-case-19d09f653c98, 2016.

[154] M. P. Sampat, Z. Wang, S. Gupta, A. C. Bovik, and M. K. Markey, "Complex wavelet structural similarity: A new image similarity index," *IEEE transactions on image processing*, 2009.

[155] G. Santos, K. V. Paixao, N. Anquetil, A. Etien, M. de Almeida Maia, and S. Ducasse, "Recommending source code locations for system specific transformations," in *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, 2017, pp. 160–170.

[156] SeleniumHQ, *Selenium ide*, `http://docs.seleniumhq.org/projects/ide`, 2006.

[157] K. Sen, S. Kalasapur, T. Brutch, and S. Gibbs, "Jalangi: A selective record-replay and dynamic analysis framework for javascript," in *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, 2013.

[158] A. Sharma, *8 quick tips to speed up android app development*, `https://appinventiv.com/blog/8-quick-tips-speed-android-app-development`, 2018.

[159] S. Sidiroglou-Douskos, E. Lahtinen, A. Eden, F. Long, and M. Rinard, "Codecarboncopy," in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ser. ESEC/FSE 2017, New York, NY, USA: ACM, 2017, pp. 95–105.

[160] S. Sidiroglou-Douskos, E. Lahtinen, F. Long, and M. Rinard, "Automatic error elimination by horizontal code transfer across multiple applications," in *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '15, New York, NY, USA: ACM, 2015, pp. 43–54.

[161] J. Steven, P. Chandra, B. Fleck, and A. Podgurski, "Jrapture: A capture/replay tool for observation-based testing," in *Proceedings of the International Symposium on Software Testing and Analysis*, 2000.

[162] B. Tabbaa, *Small is beautiful the big bang launch failure of healthcare.gov*, `https://medium.com/@bishr_tabbaa/small-is-beautiful-the-launch-failure-of-healthcare-gov-5e60f20eb967`, 2018.

[163] S. H. Tan, Z. Dong, X. Gao, and A. Roychoudhury, "Repairing crashes in android apps," in *Proceedings of the 40th International Conference on Software Engineering*, 2018.

[164] G. Tassey, "The economic impacts of inadequate infrastructure for software testing," *National Institute of Standards and Technology*, 2002.

[165] S. Thummalapenta, S. Sinha, N. Singhania, and S. Chandra, "Automating test automation," in *Proceedings of the 34th International Conference on Software Engineering*, 2012.

[166] F. Thung, X. D. Le, D. Lo, and J. Lawall, "Recommending code changes for automatic backporting of linux device drivers," in *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, 2016, pp. 222–232.

[167] Tricentis, *Real life examples of software development failures*, `https://www.tricentis.com/softwaretesting/real-life-examples-software-development-failures`, 2019.

[168] D. Vadas and J. R. Curran, "Programming with unrestricted natural language," in *Proceedings of the Australasian Language Technology Workshop*, 2005.

[169] W3C, *Xml path language*, `https://www.w3.org/TR/xpath-30`, 2014.

[170] U. of Waikato, *Weka*, `http://www.cs.waikato.ac.nz/ml/weka`, 1997.

[171] L. Wei, Y. Liu, and S.-C. Cheung, "Taming android fragmentation: Characterizing and detecting compatibility issues for android apps," in *2016 IEEE/ACM International Conference on Automated Software Engineering (ASE)*, 2016.

[172] W. Weimer, T. Nguyen, C. Le Goues, and S. Forrest, "Automatically finding patches using genetic programming," in *Proceedings of the 31st International Conference on Software Engineering*, Washington, DC, USA: IEEE Computer Society, 2009, pp. 364–374.

[173] WordPress, *Wordpress*, `https://play.google.com/store/apps/details?id=org.wordpress.android`, 2018.

[174] L. Wu, M. Grace, Y. Zhou, C. Wu, and X. Jiang, "The impact of vendor customizations on android security," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, 2013.

[175] Q. Xin and S. P. Reiss, "Leveraging syntax-related code for automated program repair," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017.

[176] Z. Xing and E. Stroulia, "Api-evolution support with diff-catchup," *IEEE Transactions on Software Engineering*, pp. 818–836, 2007.

[177] J. Xuan, M. Martinez, F. DeMarco, M. Clement, S. L. Marcote, T. Durieux, D. L. Berre, and M. Monperrus, "Nopol: Automatic repair of conditional statement bugs in java programs," *IEEE Transactions on Software Engineering*, 2017.

[178] G. Yang, J. Jones, A. Moninger, and M. Che, "How do android operating system updates impact apps?" In *Proceedings of the 5th International Conference on Mobile Software Engineering and Systems*, 2018.

[179] W. Yang, M. R. Prasad, and T. Xie, "A grey-box approach for automated gui-model generation of mobile applications," in *Proceedings of the 16th International Conference on Fundamental Approaches to Software Engineering*, 2013.

[180] T. Yeh, T.-H. Chang, and R. C. Miller, "Sikuli: Using gui screenshots for search and automation," in *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology*, 2009.

[181] H. Zadgaonkar, *Robotium Automated Testing for Android*. Packt Publishing Ltd, 2013.

[182] A. Zeller and R. Hildebrandt, "Simplifying and isolating failure-inducing input," *IEEE Transactions on Software Engineering*, 2002.

[183] X. Zhou, Y. Lee, N. Zhang, M. Naveed, and X. Wang, "The peril of fragmentation: Security hazards in android device driver customizations," in *2014 IEEE Symposium on Security and Privacy (SP)*, 2014.

[184] M. H. Y. Zhu, R. Peri, and V. J. Reddi, "Mosaic: Cross-platform user-interaction record and replay for the fragmented android ecosystem," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, 2015.

[185] D. Zuddas, W. Jin, F. Pastore, L. Mariani, and A. Orso, "Mimic: Locating and understanding bugs by analyzing mimicked executions," in *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering (ASE)*, 2014.